Opérateurs arithmétiques matériels pour des applications spécifiques

Nicolas Veyrat-Charvillon

N° d'ordre : 411 N° attribué par la bibliothèque : 07ENSL0 411

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée et soutenue publiquement le 28 juin 2007 par

Nicolas VEYRAT-CHARVILLON

pour l'obtention du grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l'École doctorale de mathématiques et d'informatique fondamentale de Lyon

Opérateurs arithmétiques matériels pour des applications spécifiques

Directeurs de thèse : Jean-Michel MULLER
Arnaud TISSERAND

Après avis de : Tomas LANG
Philippe LANGLOIS

Devant la commission d'examen formée de :

Daniel ETIEMBLE	Membre
Philippe LANGLOIS	Membre/Rapporteur
Peter MARKSTEIN	Membre
Jean-Michel MULLER	Membre
Alberto NANNARELLI	Membre
Arnaud TISSERAND	Membre

Remerciements

Je tiens à remercier mes directeurs de thèse, Arnaud Tisserand pour m'avoir offert la chance de faire ma thèse puis m'avoir supporté pendant tout ce temps, Jean-Michel pour sa patience et ses précieux conseils.

Je remercie mes relecteurs Thomás Lang et Philippe Langlois. Merci au président du jury Daniel Étiemble, et à ses autres membres Peter Markstein et Alberto Nannarelli qui sont venus de loin.

Je remercie aussi Nicolas Brisebarre, Florent de Dinechin, Laurent Imbert, Claude-Pierre Jeannerod, Nathalie Revol, Serge Torres et Gilles Villard pour leur gentillesse et leurs réponses à des questions (parfois ?) stupides.

Je tiens à remercier mes cobureaux Guillaume, Sylvie, Jérémie, Saurabh, Romain, Ilia, Florent, Guillaume (l'autre) et Francisco pour l'unique atmosphère de «travail productif» qu'ensemble nous avons réussi à créer.

Structure fondamentale du LIP, il me faut aussi remercier le coin café, et ceux qu'on a le plaisir d'y rencontrer : tous ceux déjà cités, nos secrétaires Sylvie, Corinne, Isabelle, Edwige et Danielle, les autres membres d'Arénaire : Nicolas, Pascal, Édouard, Damien, Sylvain, Christoph et Vincent, ou du laboratoire : Dominique, Eddy, Yves et plein d'autres.

Merci aux sportifs Saurabh et Jean-Luc, et aux testeurs de cartes graphiques : Jérémie, Guillaume, Damien, Florent, Victor, Vincent, et même Arnaud et Nicolas.

Merci tout particulièrement à Jérémie pour plein de trucs, à Jean-Luc pour les même raisons, à Saurabh «Ghandu» pour son vacabulaire et une magnifique kurta, à Ryan, Gabe et Martha pour Thanksgiving, et enfin à Romain pour tout.

Pour finir, je remercie de tous cœur mes parents Jean-Claude et Marie-Line qui m'ont accompagné depuis le début, qui le font encore, de même que mon frère Mathieu et ma sœur Marion. Merci à Martine, Jean-Mi, Bérangère, Philippe et Isabelle pour être venu me soutenir, et à toute ma famille qui l'a fait par écrit.

Table des matières

Liste des figures

Liste des tables

Introduction

Les avancées de ces dernières années en architecture des ordinateurs, en micro-électronique et dans les technologies de fabrication ont permis aux performances des circuits intégrés numériques de continuer leur progression exponentielle. En parallèle, on constate une explosion du coût du développement et de la fabrication de circuits intégrés, coût qui devrait continuer à augmenter dans un avenir proche. Il devient alors important de concevoir des unités de calcul qui seront réutilisables dans plusieurs applications et sur plusieurs technologies.

Depuis quelques années, l'industrie des circuits intégrés utilise de plus en plus des cœurs et des blocs IP (*Intellectual Property*). Ces cœurs ou blocs sont des descriptions de diverses fonctionnalités qui peuvent être implantées sur différentes cibles. Leur conception est devenue une industrie à part entière. Toutefois, seuls des blocs avec des fonctionnalités figées et peu variées sont disponibles actuellement. Par exemple, on ne trouve pas de bloc capable de faire des multiplications, divisions et racines carrées dans une même portion de circuit.

L'arithmétique des ordinateurs est une branche de l'informatique qui traite de l'étude et de la conception des moyens permettant d'effectuer les calculs de base en machine [11, 13]. Ceci recouvre l'implantation d'unités de calcul matérielles, comme les unités arithmétiques et logiques, d'opérateurs spécifiques, par exemple des filtres pour le traitement du signal, ou de bibliothèques logicielles pour l'évaluation des fonctions élémentaires [12].

L'arithmétique des ordinateurs comporte trois aspects fondamentaux :

- *les systèmes de représentation des nombres*, ou la manière dont les valeurs mathématiques sont codées en machine. Quelques formats de représentation couramment utilisés sont décrits plus loin.
- *les algorithmes de calcul* pour l'évaluation des fonctions arithmétiques. Ces fonctions vont des opérations de base comme l'addition et la multiplication à l'évaluation de fonctions plus complexes, comme la racine carrée ou les fonctions trigonométriques.
- *leur implantation* matérielle et logicielle. Celle-ci couvre l'optimisation des performances (en vitesse, en surface de circuit, en consommation d'énergie, respect des contraintes temps réel), mais aussi le test et la validation d'opérateurs.

L'objet de cette thèse est la conception, tant théorique que pratique, d'opérateurs de calcul en matériel. Les domaines d'application visés pour ces opérateurs arithmétiques sont le traitement numérique du signal et des images ainsi que la cryptographie. Elle répond à un besoin croissant en support matériel pour des algorithmes évolués et nécessitant de nombreuses fonctionnalités élémentaires. Ce thème de recherche comporte à la fois des aspects mathématiques et informatiques. Il porte plus précisément sur l'étude et l'implantation matérielle d'opérateurs arithmétiques pour des applications spécifiques. La spécialisation de nos opérateurs arithmétiques va permettre d'offrir des unités de calcul évoluées et performantes sur une faible surface de silicium. Via l'écriture de générateurs automatiques, nous somme capables de fournir facilement des descriptions matérielles

d'opérateurs optimisés pour différentes applications.

Représentation des nombres

La valeur mathématique v d'un nombre entier ou réel approché est codée à l'aide d'un ensemble de n bits notés $\{v_{n-1}, \cdots, v_0\}$. Dans cette thèse, on s'intéresse à des nombres entiers ou virgule fixe (des nombres fractionnaires binaires). Pour lire v, on établit une bijection entre les valeurs booléennes codées électriquement par chaque fil, et l'ensemble de chiffres $\{0, 1\}$ pour chacun des v_i. Il existe différentes manières d'interpréter la valeur de ces fils pour en déduire la valeur mathématique v.

Les formats de représentation présentés ci-dessous permettent de coder des entiers. Si on veut travailler sur des approximations de réels, on introduit un facteur d'échelle $\text{LSB}(v)$ (pour *Least Significant Bit*) par lequel on multiplie l'entier codé pour obtenir v. Ce facteur est une puissance de 2 constante pour une notation donnée, qui indique le poids du bit le plus faible. Par exemple, le $\text{LSB}(v)$ vaut 1 si on travaille sur des entiers. Si $\text{LSB}(v) = 2^{-2}$, alors on sait que les valeurs v ont deux bits après la virgule. On est ainsi capable d'attribuer une valeur entière ou virgule fixe à un vecteur de valeurs logiques codées électriquement.

– *La notation non signée* représente une valeur positive v par :

$$v = \text{LSB}(v) \sum_{i=0}^{n-1} v_i \, 2^i,$$
$$\text{d'où } v \in [0, 2^n - 1] \times \text{LSB}(v).$$

– *La notation signe-valeur absolue* est une extension directe de la notation non signée aux nombres relatifs. On adjoint simplement en tête un bit représentant le signe du nombre à la représentation non signée de sa valeur absolue. Une valeur v est codée par :

$$v = (-1)^{v_{n-1}} \times \text{LSB}(v) \sum_{i=0}^{n-2} v_i \, 2^i, \text{ où } \left\{ \begin{array}{l} v \geq 0 \text{ si } v_{n-1} = 0 \\ v \leq 0 \text{ si } v_{n-1} = 1 \end{array} \right. ,$$
$$\text{d'où } v \in [-2^{n-1} + 1, 2^{n-1} - 1] \times \text{LSB}(v).$$

Cette notation possède une double représentation de la valeur 0 ($+0$ et -0).

– *Le complément à 2* est une autre notation des nombres relatifs. Une valeur v est codée en complément à 2 par :

$$v = \text{LSB}(v) \left(-2^{n-1} \, v_{n-1} + \sum_{i=0}^{n-2} v_i \, 2^i \right),$$
$$\text{d'où } v \in [-2^{n-1}, 2^{n-1} - 1] \times \text{LSB}(v).$$

Bien que moins intuitif que la notation signe-valeur absolue, le complément à 2 résout un problème inhérent à l'écriture précédente. Pour pouvoir additionner deux nombres relatifs en signe-valeur absolue, il faudra un algorithme d'addition si les deux nombres sont de même signe, et un algorithme de soustraction sinon. Dans la représentation en complément à 2, l'addition reste la même quels que soient les signes des opérandes. D'autre part, les additionneurs et soustracteurs y sont similaires. Par contre, passer d'un nombre à son opposé est plus compliqué.

– *La notation à retenue conservée* (ou *Carry Save*) est surtout utilisée dans les calculs intermédiaires. Il s'agit d'un système de représentation redondant [11], qui utilise $2 \times n$ bits notés $\{c_{n-1}, \cdots, c_0\}$ et $\{s_{n-1}, \cdots, s_0\}$. On travaille toujours en base 2, mais on a un ensemble de chiffres $\{0, 1, 2\}$, en posant $v_i = c_i + s_i$ pour chaque chiffre de la valeur $v = c + s$. Les nombres c et s peuvent être en non signé ou en complément à 2. Par exemple, l'extension à retenue conservée du complément à 2 est :

$$v = \text{LSB}(v) \left(-2^{n-1} \, (c_{n-1} + s_{n-1}) + \sum_{i=0}^{n-2} (c_i + s_i) \, 2^i \right),$$
$$\text{d'où } v \in [-2^n, 2^n - 2] \times \text{LSB}(v).$$

Cette notation présente l'avantage suivant : il est possible d'additionner *en parallèle* un nombre en retenue conservée et un nombre en notation simple de position (ou un autre nombre en retenue conservée), et d'obtenir leur somme en retenue conservée. Par contre, la notation à retenue conservée utilise deux fois plus de fils que les notations simples de position, ce qui implique un routage et une mémorisation deux fois plus coûteux, et crée des opérateurs potentiellement plus gros.

– *La notation Borrow Save* est une autre notation binaire redondante proche de la notation à retenue conservée, mais avec des chiffres dans l'ensemble $\{-1, 0, +1\}$. On code cette fois un chiffre v_i par soustraction de deux bits : $v_i = s_i - b_i$. Par la suite, pour éviter les confusions entre les chiffres négatifs et la soustraction, on note les chiffres négatifs en les surmontant d'une barre. Par exemple, $-5 = \overline{5}$. L'extension borrow-save du non signé est :

$$v = \text{LSB}(v) \sum_{i=0}^{n-1} (s_i - b_i) \, 2^i,$$
$$\text{d'où } v \in [-2^n - 1, 2^n - 1] \times \text{LSB}(v).$$

Un des intérêts de cette notation par rapport à celle à retenue conservée est la manipulation plus naturelle des nombres négatifs. En particulier, calculer l'opposé d'un nombre est trivial, puisqu'il suffit de permuter les bits s_i et b_i.

Le format virgule fixe est celui qui est le plus couramment utilisé dans les applications du traitement du signal que visent nos opérateurs. Dans le cas des applications cryptographiques et du traitement des images, on travaille le plus souvent en arithmétique entière.

Un autre format, plus courant dans les processeurs généralistes, est le format virgule flottante. Il permet de représenter des valeurs avec une plus grande dynamique, au prix d'opérateurs plus complexes. Nous n'utiliserons pas ce format dans cette thèse.

Exemple 0.1 *Soit v codée par les 4 bits $(v_3, v_2, v_1, v_0) = (1, 1, 0, 1)$. La valeur v dépend de la notation :*

– *Si on veut coder des valeurs entières positives, le chiffre de poids faible v_0 doit avoir un poids de 1. On pose donc $\text{LSB}(v) = 1$, et on lit $v = 13$.*
– *Par contre, si on travaille dans l'intervalle $[0, 1[$, les v doivent vérifier $|v| \leq 1$. On pose donc $\text{LSB}(v) = 2^{-4}$ pour un codage non signé, et on lit $v = 13 \times 2^{-4} = \frac{13}{16} = 0.8125$.*
– *Si on travaille dans l'intervalle $[-1, 1[$ en complément à 2, on pose $\text{LSB}(v) = 2^{-4+1}$ (on utilise un fil pour le signe). On lit alors $v = (-2^3 + 5) \times 2^{-3} = -\frac{3}{8} = -0.375$.*

Algorithmes de calcul

L'évaluation en matériel de fonctions numériques est un point crucial dans de nombreuses applications. L'inverse et la fonction sinus sont souvent utilisées en traitement du signal. Les fonctions transcendantes sont récurrentes en calcul flottant, et des fonctions comme la racine carrée et la norme euclidienne se retrouvent souvent dans des applications graphiques. Le calcul de ces fonctions dans des opérateurs spécialisés est souvent beaucoup plus efficace que l'utilisation de structures génériques (additionneurs et multiplieurs). De nombreuses méthodes existent pour l'évaluation de fonctions en matériel : méthodes à base de tables, algorithmes à récurrence de chiffres, approximations polynomiales ou rationnelles, ou encore des combinaisons de ces solutions.

Les méthodes à base de tables sont souvent utilisées pour des applications avec de petits besoins en précision. Des précisions plus grandes peuvent être obtenues en combinant tables et opérations arithmétiques [16, 26].

Les algorithmes à récurrence de chiffre, ou algorithmes décalage-addition, produisent un chiffre du résultat par itération en partant du chiffre de poids le plus fort, comme la division à la main sur papier. Les deux plus fameux algorithmes à récurrence de chiffre sont : les algorithmes de type SRT pour la division, racine carrée et autres fonctions algébriques [11], et l'algorithme CORDIC pour les fonctions élémentaires [12]. Ces algorithmes utilisent seulement des additions et décalages (plus éventuellement des petites tables) pour le calcul de chaque itération. Ils donnent lieu à des opérateurs qui sont petits, mais qui ont un forte latence à cause de leur convergence linéaire. Augmenter leur base de travail réduit le nombre d'itérations nécessaires, mais demande des circuits sensiblement plus gros.

Les approximations polynomiales sont largement utilisées pour l'évaluation de fonctions, même en matériel où la taille des multiplieurs utilisés pour leur évaluation est un problème majeur. Ces approximations permettent d'évaluer la plupart des fonctions courantes pour peu qu'elles soient régulières. Les approximations rationnelles permettent une évaluation plus précise, mais elles requièrent une division qui ajoute au coût matériel et introduit une latence supplémentaire souvent importante.

Implantation d'opérateurs matériels

Le but d'une implantation matérielle des fonctions numériques est l'amélioration des performances. Par exemple, l'implantation de fonctions cryptographiques en matériel fait habituellement gagner plusieurs ordres de grandeur en vitesse par rapport aux mêmes fonctions codées en logiciel. En plus d'être rapide, un opérateur doit aussi occuper la plus petite surface possible pour des raisons de coût.

Maintenir une bibliothèque contenant les descriptions en VHDL d'opérateurs spécialisés est difficile, du fait de la gestion limitée des paramètres génériques dans ce langage. Notre approche est donc, plutôt que de fournir une bibliothèque VHDL, d'écrire des programmes en C ou C++ qui génèrent des descriptions d'opérateurs optimisés. L'utilisation de générateurs permet d'effectuer des calculs plus complexes que ceux qu'on ferait à l'aide de formules VHDL, et d'explorer l'espace des paramètres. On trouve ainsi des solutions bien plus efficaces que ce qu'on peut obtenir à l'aide du VHDL seul. La rigidité du VHDL rend aussi difficile l'écriture de certaines structures, même si elles sont régulières, comme par exemple des arbres.

Les implantations matérielles des opérateurs décrits dans cette thèse ont été faites sur des circuits de type FPGA (pour *Field-Programmable Gate Array*). Bien qu'ils soient moins performants que des circuits numériques spécialisés de type ASIC (pour *Application-Specific Integrated Cir-*

cuits), les FPGA ont un coût de développement bien moindre et leur prototypage est beaucoup plus facile et rapide. De plus, ils disposent de ressources suffisantes même pour des architectures complexes. Toutes ces raisons en font de bons candidats pour la mise en pratique de nos travaux.

Les résultats présentés dans cette thèse peuvent être facilement adaptés aux circuits ASIC, moyennant quelques modifications pour adapter les opérateurs à la structure particulière des ASIC. Par exemple, l'absence de lignes de *fast-carry* demanderait l'utilisation d'un algorithme plus efficace que l'addition à propagation de retenue, comme des additionneurs à saut de retenue (*carry-skip*) [11].

Travaux réalisés

Cette thèse étudie des opérateurs arithmétiques matériels dédiés à l'évaluation de fonctions spécifiques. La plupart des travaux présentés ont été réalisés en collaboration avec mon directeur de thèse Arnaud Tisserand, et avec Romain Michard, un autre étudiant en thèse dans l'équipe Arénaire. Ce document se divise en trois chapitres. Une méthode pour l'évaluation de fonctions basée sur des approximations polynomiales calculées par des petits opérateurs est présentée dans le premier chapitre. Le deuxième chapitre décrit les résultats que nous avons obtenus sur la génération automatique d'opérateurs à récurrence de chiffres optimisés. Ces opérateurs sont utilisés pour le calcul de certaines fonctions algébriques, dont la division et la racine carrée. Le troisième chapitre présente une implantation performante d'opérateurs pour des fonctions de hachage cryptographique. Nous y décrivons aussi une architecture où toute une famille de fonctions est calculée dans le même composant, avec un partage très efficace des ressources.

Les approximations polynomiales et rationnelles sont un moyen efficace d'approcher des fonctions mathématiques. Nous avons travaillé sur des petits opérateurs capables d'évaluer un ensemble de fonctions à l'aide d'approximations polynomiales. Un algorithme implanté en C++ fournit des approximations avec des coefficients creux et où les puissances de l'argument d'entrée sont tronquées pour minimiser leur coût matériel. Un grande partie du matériel est partagée entre l'évaluation des différentes fonctions, ce qui permet d'en faire une solution bien plus intéressante qu'une implantation séparée des opérateurs. Cette méthode est l'objet du premier chapitre.

Il est possible de calculer efficacement des fonctions algébriques simples comme la division, la racine carré et la racine carrée inverse, voire la norme euclidienne, à l'aide des opérateurs à additions et décalages. Ces opérateurs sont efficaces et demandent très peu de matériel, mais leur mise au point résulte d'un processus complexe et est sujette à erreurs au niveau mathématique. Ils sont aussi difficiles à traduire en circuits. Les travaux qui traitent de ces opérateurs laissent souvent de côté les aspects pratiques et l'implantation, ce qui en fait une solution peu intéressante pour les non spécialistes. Afin de faciliter leur mise en œuvre, nous proposons un logiciel, `Divgen`, qui permet d'obtenir facilement des descriptions VHDL optimisées de diviseurs à additions et décalages. Nous sommes actuellement en train d'étendre ce logiciel afin qu'il soit capable de générer des opérateurs à récurrence de chiffres pour quelques autres fonctions algébriques. Un traitement automatisé, bien que compliqué à mettre en place, autorise des améliorations qu'il est difficile, voire impossible de gérer à la main. Le programme `Divgen`, ainsi que les algorithmes proposés pour une extension vers d'autres fonctions algébriques, sont présentés dans le second chapitre.

Le coût matériel d'une architecture partagée pour des fonctions cryptographiques proches peut atteindre une performance comparable à celle d'implantations séparées, pour un coût matériel très inférieur. Dans le cadre d'une collaboration avec le laboratoire ATIPS de l'Université de Calgary, nous avons travaillé sur une implantation de la famille de fonctions de hachage cryptographique

SHA-2. Nous avons réalisé une implantation améliorée de ces fonctions, avec des besoins matériel réduits. Nous proposons aussi une version fonctionnant à haute fréquence. Finalement nous avons obtenu une architecture multifonction capable d'effectuer soit un hachage SHA-384 ou SHA-512, ou bien de se comporter comme deux opérateurs SHA-224 ou SHA-256 indépendants. Cette capacité augmente la flexibilité pour des applications allant de serveurs calculant en parallèle pour des flux multiples à la génération pseudo-aléatoire de nombres. Les travaux réalisés sont décrits dans le troisième chapitre.

Dans le cadre d'une collaboration avec Romain Michard, j'ai aussi étudié la E-méthode due à Ercegovac [10]. Cette méthode permet d'évaluer efficacement des approximations rationnelles de fonctions en évitant d'effectuer la division finale dans un opérateur séparé. Nous avons proposé une implantation efficace de cette méthode lors de du *10ème SYMPosium en Architectures nouvelles de machines (SYMPA)* en 2005 [1]. Une version pour la basse consommation a été présentée aux *5ème journées d'études Faible Tension Faible Consommation (FTFC)* la même année [2]. Nous n'avons pas donné suite à ces travaux, je ne les traiterai donc pas dans ce document.

CHAPITRE 1

Opérateurs matériels pour l'évaluation de fonctions multiples

Un algorithme pour la génération d'opérateurs arithmétiques matériels dédiés à l'évaluation de fonctions multiples est présenté. Il fournit des approximations polynomiales avec des coefficients creux et où les puissances sont estimées. Les multiplications exactes y sont remplacées par des opérations approchées moins coûteuses à implanter, ou par de courtes séquences d'additions/soustractions. Les puissances « tronquées » sont partagées entre les différentes fonctions. Notre algorithme a été implanté sous la forme d'un programme C++ qui génère une description VHDL optimisée des opérateurs. Quelques exemples ont été synthétisés sur FPGA et comparés à d'autres méthodes d'évaluation de fonctions. Notre algorithme permet d'obtenir des opérateurs efficaces avec une erreur d'évaluation moyenne faible pour des applications en traitement du signal et des images, par exemple dans des filtres.

Une première version de ces travaux, pour des opérateurs dédiés à l'évaluation d'une seule fonction a été présentée lors de la *16ème International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, à Samos en 2005 [3]. Cet article y avait remporté le prix du meilleur papier. Une version journal de l'extension présentée dans ce chapitre est en cours de finalisation.

1.1 Introduction

Les opérateurs arithmétiques de base (pour l'addition/soustraction, la multiplication, la division et la racine carrée) ont très souvent fait l'objet d'implantations matérielles performantes dans les circuits numériques [11]. Des applications récentes demandent l'évaluation rapide d'opérations plus complexes telles la racine carrée inverse, les fonctions trigonométriques, le logarithme, l'exponentielle et des composées de ces fonctions.

Les approximations polynomiales sont souvent utilisées pour l'évaluation de fonctions dans des implantations tant logicielles que matérielles. Les fonctions élémentaires comme le sinus, cosinus, l'exponentielle ou le logarithme sont souvent évaluées à l'aide de polynômes [12]. Les fonctions algébriques comme la racine carrée ou la racine carrée inverse peuvent aussi s'évaluer efficacement en utilisant des polynômes. Des polynômes de petit degré sont souvent utilisés pour calculer des inverses dans les applications du traitement du signal, par exemple la démodulation de fréquence.

Ce chapitre présente un algorithme pour la génération automatique d'opérateurs arithmétiques matériels dédiés à l'évaluation de fonctions multiples. Il fournit des approximations polynomiales

avec des coefficients « creux » et des puissances « tronquées ». Les coefficients dit creux sont composés de seulement quelques chiffres non nuls, et permettent le remplacement de certaines multiplications dans l'évaluation du polynôme par de courtes séquences d'additions/soustractions. Les puissances dites tronquées remplacent les multiplications exactes par des opérations approchées bien plus petites à implanter. Elles sont calculées par du matériel commun à toutes les fonctions évaluées. L'algorithme proposé ici donne des approximations avec une erreur moyenne très petite et une erreur maximum limitée à quelques LSB(Least Significant Bits). Ceci fait de nos opérateurs une solution intéressante pour les applications en multimédia et en traitement du signal.

Ce chapitre est organisé de la façon suivante. Les notations utiles et quelques jalons concernant l'évaluation de fonctions à l'aide d'approximations polynomiales sont présentés dans la section 1.2. La section 1.3 récapitule les principales méthodes existantes pour l'évaluation de fonctions en matériel. Notre algorithme est décrit en détail dans les sections 1.4 et 1.5 respectivement pour plusieurs fonctions et dans le cadre de l'étude d'une seule fonction. La section 1.6 présente les résultats obtenus par le programme C++ et les implantations FPGA des opérateurs. Ces résultats sont comparés à d'autres méthodes existantes dans la section 1.7. La section 1.8 conclut l'étude. Enfin, la section 1.9 détaille les travaux que nous avons effectué sur le calcul de puissances entières en matériel, dont les résultats sont utilisés dans ce chapitre.

Les opérateurs de [3] ne fournissent chacun une approximation que pour une seule fonction et sont limités en pratique à 16 bits de précision. Dans ce chapitre, des opérateurs pouvant approcher des fonctions multiples sont présentés. En utilisant un schéma de sélection plus complexe pour les coefficients creux (Sec. 1.5.1), des précisions jusqu'à 24 bits sont atteintes. Ce chapitre utilise aussi des schémas de troncature plus fins pour les x^i (Sec. 1.5.2). Toutes ces améliorations contribuent à des opérateurs plus efficaces, à la fois plus précis et plus petits.

1.2 Notations et notions utiles

La *fonction f approchée* a ses entrées et sorties représentées en format virgule fixe binaire (non signé ou en complément à 2). L'*argument* x est considéré exact dans le *domaine* virgule fixe $[a, b[$ et le résultat $f(x)$ est dans l'*intervalle* $[a', b'[$ ou $]a', b']$. L'extension vers d'autres types d'intervalles, par exemple $[a, b]$, est immédiate. L'argument x est un nombre de w_I bits et le résultat $f(x)$ est un nombre de w_O bits. Le nombre de bits entiers est fixé par le format de la plus petite représentation qui inclut à la fois a et b. Le nombre de bits fractionnaires est calculé par l'algorithme pour correspondre à la *précision requise* en sortie.

Un polynôme p de degré d est noté $p(x) = p_0 + p_1x + p_2x^2 + \ldots + p_dx^d$. La notation $()_2$ indique la représentation binaire standard d'une valeur, par exemple $3.125 = (11.001)_2$. La notation $()_{\text{bs}}$ indique la représentation *borrow-save* [11]. C'est une représentation binaire redondante sur l'ensemble de chiffres $\{-1, 0, 1\}$. Elle peut permettre entre autres d'éviter les longues chaînes de '1' (via un recodage de Booth modifié), par exemple $31.0 = (11111.0)_2 = (10000\overline{1})_{\text{bs}}$ (le bit négatif -1 est noté $\overline{1}$). Les multiplications par des puissances de 2 se réduisent à un simple décalage des chiffres. On note $q(x)$ le polynôme d'approximation avec des coefficient « creux », que l'on dérive du polynôme minimax. Chaque *coefficient quantifié* q_i du polynôme d'approximation $q(x)$ est composé de la somme d'au plus n_i chiffres décalés signés non nuls dans l'ensemble $\{-1, 1\}$. Le coefficient q_0 de degré 0 n'est pas quantifié.

L'*erreur d'approximation* ε_{th} mesure la différence entre la fonction mathématique f et son approximation, ici un polynôme p. On définit ε_{th} par :

$$\varepsilon_{th} = \|f - p\|_\infty = \max_{a \le x \le b} |f(x) - p(x)|. \tag{1.1}$$

L'erreur ε_{th} est estimée numériquement par la fonction `infnorm` de Maple. ε_{th} est la valeur théorique *minimale* de l'erreur qu'on peut obtenir en approchant f à l'aide du polynôme p. À cause de la précision finie des coefficients et des calculs intermédiaires lors de l'évaluation de p, on rencontre en pratique des erreurs supplémentaires. Les *erreurs d'arrondi* qui résultent de la nature discrète des résultats intermédiaires et finaux s'accumulent et se rajoutent à l'erreur d'approximation. Cette erreur reste petite pour une seule opération, au plus le poids du LSB. Cependant, dans le cadre d'une suite d'opérations, ces petites erreurs peuvent s'accumuler et diminuer la précision du résultat final de façon significative. Afin de limiter les effets des erreurs d'arrondi, on introduit g *bits de garde*. Les calculs intermédiaires sont donc faits sur $w_O + g$ bits.

Les approximations polynomiales utilisées par la suite sont dérivées de l'approximation polynomiale *minimax*. Le polynôme d'approximation minimax de degré d pour la fonction f sur $[a, b]$ est le polynôme p^* qui vérifie $\|f - p^*\|_\infty = \min_{p \in \mathcal{P}_d} \|f - p\|_\infty$, où $\mathcal{P}_d$ est l'ensemble des polynômes à coefficients réels et de degré au plus d. Les approximations minimax peuvent être calculées grâce à un algorithme dû à Rémès [19]. On en trouve une bonne description dans [12]. Les polynômes minimax de ce chapitre sont calculés numériquement à l'aide de la fonction `minimax` de Maple.

Les erreurs sont exprimées directement ou en précision équivalente. La *précision* est le nombre de bits corrects ou significatifs. L'erreur absolue ε et la précision ν sont reliées par la relation $\nu = -\log_2 |\varepsilon|$ pour le format binaire pur, c'est à dire les valeurs représentées sous la forme $(\pm 0.x_1 x_2 x_3 \ldots x_w)_2$. Par exemple, une erreur $\varepsilon = 0.0000107$ est équivalente à une précision $\nu = 16.5$ bits.

Le résultat de l'approximation polynomiale calculée par l'opérateur matériel, avec des coefficients quantifiés, des puissances estimées et des calculs de précision finie, est noté $\tilde{q}(x)$. Pour un argument donné x il est possible d'évaluer l'erreur totale $\tilde{\varepsilon}(x) = f(x) - \tilde{q}(x)$ qui inclut tous les types d'erreurs. Dans la mesure où notre méthode vise des précisions modérées (jusqu'à 24 bits en entrée), l'*erreur moyenne* totale ε_{avg}, son *écart-type* σ et l'*erreur maximum* ε_{max} peuvent être calculées. En effet, pour toutes les valeurs possibles de x, le résultat $\tilde{q}(x)$ tel que fourni par l'opérateur peut être calculé et comparé à la valeur théorique $f(x)$. La précision visée est notée μ, c'est à dire que nos opérateurs doivent fournir un $\varepsilon_{\text{avg}} \le \mu$.

Au cours de ce chapitre, un exemple récurrent est utilisé pour illustrer le fonctionnement de notre algorithme. Il s'agit d'une étude de la fonction sinus ($f(x) = \sin(x)$) sur le domaine $[a, b[= [0, \frac{\pi}{4}[$. L'approximation polynomiale a un degré de 3, et la précision moyenne visée est 12 bits ($\mu = 2^{-12}$).

1.3 Travaux précédents

De nombreuses méthodes ont été proposées pour l'évaluation de fonctions en matériel : méthodes à base de tables, algorithmes à récurrence de chiffre, approximations polynomiales ou rationnelles, ou encore des combinaisons de ces solutions. Voir [11], [12] et [24] pour plus de détails.

Les méthodes à base de tables sont souvent utilisées pour des applications avec de petits besoins en précision. Des précisions plus grandes peuvent être obtenues en combinant tables et opérations arithmétiques : tables et additions [16], tables et petites multiplications [21, 26]. Ces méthodes donnent des implantations efficaces pour des FPGA à base de tables look-up. Un de leurs inconvénients est qu'elle sont dédiées à l'évaluation d'une seule fonction.

Les approximations polynomiales sont largement utilisées pour l'évaluation de fonctions [12], même en matériel [17, 15] où la taille des multiplieurs est un problème majeur. Plusieurs solutions ont été étudiées pour diminuer leur taille. Dans [32] une méthode basée sur un polynôme d'approximation de degré 2 et une unité d'élévation au carré (squarer) spécialisée est proposée. Cette solution réduit la surface de moitié par rapport aux méthodes habituelles. Nous verrons plus loin que le choix d'un polynôme qui ait des coefficients exactement représentables dans le format cible est un des principaux problèmes rencontrés lors de l'implantation d'approximations polynomiales. Une méthode générale mais très lente pour les hauts degrés qui permet de résoudre ce problème est proposée dans [14]. Les approximations polynomiales peuvent aussi être combinées à des tables. Dans [18] une méthode basée sur des réductions d'arguments et des expansions de séries est utilisée pour le calcul d'inverses, de racines carrées, racines carrées inverses et de quelques fonctions élémentaires à l'aide de petits multiplieurs et tables. Une méthode récente basée sur des polynômes et tables est proposée dans [27].

La méthode des tableaux de produits partiels [28] utilise des séries convergentes où toutes les opérations ont été développées jusqu'au niveau du bit et où les termes de poids faible sont négligés. La méthode des sommes pondérées utilise une idée similaire avec des polynômes [28, 29]. Les coefficients du polynôme sont distribués au niveau du bit pour récrire le polynôme comme la somme d'un immense ensemble de produits pondérés des bits de x. Une partie de ces termes est ensuite éliminée. Cette méthode donne des résultats intéressants pour plusieurs fonctions.

1.4 Algorithme pour un opérateur évaluant plusieurs fonctions

La description de notre algorithme est donc découpée en deux parties. Dans cette section nous décrivons la façon dont plusieurs fonctions peuvent être calculées dans un seul opérateur. Ce niveau de l'algorithme fait appel à la génération d'approximations polynomiales pour une seule fonction décrite en section 1.5.

Découpage de l'intervalle d'entrée

Lorsqu'on a besoin d'améliorer la qualité numérique d'une approximation polynomiale, on peut soit augmenter le degré du polynôme, potentiellement au coût d'un opérateur bien plus gros, ou alors découper l'intervalle d'entrée. Les améliorations de la méthode présentée dans ce chapitre permettent d'obtenir des polynômes d'un degré qui peut atteindre 6. En pratique, le coût matériel d'un opérateur de degré 6 devient prohibitif. Si la fonction ne peut toujours pas être approchée avec suffisamment de précision sur le domaine d'entrée, il faut alors découper l'intervalle d'entrée en morceaux plus petits. On génère un opérateur pour chaque sous-intervalle, puis on les regroupe en un opérateur global qui évalue la fonction sur la totalité du domaine $[a, b[$.

Le programme qui implante notre algorithme est capable de découper automatiquement l'intervalle d'entrée et de fusionner des opérateurs. Si un paramètre k est spécifié, le domaine $[a, b[$ est divisé en 2^k sous-intervalles de même taille, et un opérateur est calculé pour chacun de ces sous-intervalles. Par exemple, si on donne une valeur $k = 2$ lors de l'étude de la fonction $\frac{1}{1+x}$ où $x \in [0, 1[$, le programme va générer des opérateurs pour les intervalles $[0, \frac{1}{4}[$, $[\frac{1}{4}, \frac{1}{2}[$, $[\frac{1}{2}, \frac{3}{4}[$ et $[\frac{3}{4}, 1[$. Un opérateur fonctionnant sur $[0, 1[$ est ensuite obtenu par fusion de ces opérateurs.

1.4.1 Lecture de l'argument

Avant de générer l'opérateur, on peut effectuer une modification de l'argument. Il ne s'agit pas d'une véritable réduction d'argument [12], qui est un problème non traité dans ce chapitre, mais d'un changement de variable. On se contente de changer l'interprétation de la valeur physique de l'argument x, c'est à dire la valeur mathématique que l'on déduit des bits x_1, x_2, ..., x_{w_I}. Ce changement de format s'exprime mathématiquement par la transformation :

$$\begin{array}{l} x \leftarrow 2^{1-c}x - d, \\ \quad \text{où } c = \lceil \log_2(b-a) \rceil, \\ \quad \text{et } d = \left\{ \begin{array}{ll} 2^{-c}\,(a+2^c) & \text{si } b-a = 2^{-r}, \\ 2^{e-c}\,\lfloor 2^{-e}\,(a+2^c) \rfloor & \text{sinon.} \end{array} \right. \end{array}$$

La valeur e est calculée en prenant une valeur initiale $e = \lfloor log_2\left((b-2^c)-(a+2^c)\right) \rfloor$, puis en l'incrémentant par pas de 1 jusqu'à ce que la condition $\lfloor 2^{-e}(a+2^c) \rfloor = \lceil 2^{-e}(a+2^{c-1}) \rceil$ soit vérifiée.

Le but de cette « réduction » est d'effectuer un changement d'échelle et un décalage du domaine d'étude $[a,b[$ pour l'ajuster au mieux à l'intérieur de l'intervalle $[-1,1[$. Cette transformation permet une utilisation plus efficace de l'unité d'évaluation des puissances (Sec. 1.5.2). On commence par multiplier l'intervalle par 2^{-c}, pour lui donner une largeur aussi proche que possible de 2 (mais inférieure). Cette multiplication est transparente en matériel. On calcule ensuite un décalage d qui fasse tenir l'intervalle $[a,b[$ transformé dans $[-1,1[$, et qui soit peu coûteux. Le coût matériel de ce décalage de l'intervalle est celui de l'addition de la constante d aux bits de x, qui s'effectue sur $\max(0, 1 - \text{LSB}(d))$ bits. En pratique, ce décalage se ramène dans tous les cas étudiés à l'addition de -1, qui ne coûte qu'un inverseur.

D'une part, cette modification de l'argument permet de ne prendre en compte que les bits utiles de l'argument. Si on sait que $x \in [\frac{1}{2}, \frac{3}{4}[$, on n'a pas besoin de lire le bit de poids 2^{-1} de x. Elle introduit aussi une symétrie autour de 0 qui sert à diminuer encore le coût matériel grâce à une astuce présentée dans [20] : dans la mesure où les puissances entières sont des fonctions symétriques ou antisymétriques autour de 0, calculer $(-x)^i$ connaissant x^i est facile car

$$(-x)^i = \left\{ \begin{array}{rl} x^i & \text{si } i \text{ est pair,} \\ -(x^i) & \text{si } i \text{ est impair.} \end{array} \right.$$

En complément à 2, $-(x^i)$ peut être facilement approché avec une petite erreur (2^{-w_I-g}) en inversant les bits de x^i. Il est donc possible d'étendre le calcul des puissances de x de l'intervalle $[0,1[$ vers $[-1,1[$ au prix de quelques portes XOR (ou exclusif). Le nombre de bits de l'argument en entrée de l'unité des puissances est de $(w_I - c)$ bits seulement, ce qui permet une diminution importante de sa taille. Un bit supplémentaire est utilisé comme signe de l'argument réduit.

Exemple 1.1 *La fonction* $\sin(x)$ *avec* $x \in [0, \frac{\Pi}{4}[$ *et* $w_I = 12$ *bits subit une transformation* $x \leftarrow 2^1 x - 1$. *L'étude en est ramenée à celle de* $\sin(\frac{x+1}{2})$, *où* $x \in [-1, \frac{\pi}{2} - 1[$ *et* $w_I = (1+)11$ *(1 bit de signe et 11 bits en entrée de l'unité de puissances). Il suffit pour cela de changer la signification des bits du* $x = 0.x_1x_2x_3 \ldots x_{12}$ *original, et de les lire* $x = (-1)^{\overline{x_1}} + 0.x_2x_3 \ldots x_{12}$ *(*$\overline{x_1}$ *est le signe et* $x_2x_3 \ldots x_{12}$ *les bits de rang* -1 *à* -11 *d'une valeur en complément à 2).*
De même, l'étude de la fonction $\frac{1}{1+x}$, *où* $x \in [\frac{1}{2}, \frac{3}{4}[$ *et* $w_I = 16$ *bits est réduite à celle de* $\frac{8}{x+13}$, *où* $x \in [-1,1[$ *et* $w_I = (1+)13$ *bits seulement. On interprète* $x = (-1)^{\overline{x_3}} + 0.x_4x_5 \ldots x_{15}$.

1.4.2 Principe de la génération pour une seule fonction

La génération d'un opérateur pour l'évaluation d'une seule fonction sur l'intervalle $[a, b[$ (inclus dans $[-1, 1[$) part de l'approximation polynomiale minimax $p^*(x)$ de la fonction f. Ce polynôme à coefficients réels va subir plusieurs transformations. Leur but est d'éliminer toutes les multiplications dans l'évaluation du polynôme, et ce faisant de minimiser la taille de l'opérateur tout en limitant l'erreur introduite.

L'algorithme effectue successivement les étapes suivantes, décrites en détail dans la section 1.5.

1. Ôter les multiplications $p_i^* \times x^i$. Pour ce faire, les coefficients p_i^* sont remplacés par des valeurs q_i proches dont la représentation borrow-save ne contient qu'un petit nombre n_i de chiffres non nuls. Chaque multiplication peut alors être remplacée par $n_i - 1$ additions.

2. Éliminer les multiplications dans le calcul des puissances x^i. On part de puissances calculées exactement à l'aide de réseaux cellulaires parallèles. Une estimation $\Delta^i(x, c_i)$ est donnée à la place de x_i en tronquant le réseau du tableau de produits partiels, pour n'implanter que ses c_i colonnes de poids forts. On réalise un compromis entre l'erreur de l'estimation et le coût matériel de l'opérateur tronqué en faisant varier le nombre c_i de colonnes implantées.

3. Ajuster les coefficients q_i pour améliorer l'estimation des monômes. Ajuster aussi le coefficient q_0 de degré 0, qui n'est pas quantifié, pour améliorer la précision globale de l'opérateur.

4. Explorer le voisinage des coefficients quantifiés q_i, pour trouver d'autres solutions encore plus précises ou qui correspondent à des opérateurs plus petits.

1.4.3 Fusion d'opérateurs

Une fois qu'un opérateur a été calculé pour chaque sous-intervalle, il faut combiner les opérateurs pour obtenir un composant qui évalue la fonction sur tout le domaine de départ. Habituellement, on ajoute pour ce faire des multiplexeurs pilotés par quelques uns des MSB (Most Significant Bits) de x, et dont le rôle est de choisir le résultat du bon sous-opérateur.

Dans notre méthode, une grande partie du matériel peut être partagée entre les sous-opérateurs. Tout d'abord, l'unité de calcul des puissances peut être commune. Il suffit simplement qu'elle soit assez précise. On peut choisir pour chaque puissance x^i l'estimation $\Delta^i(x, c_i)$ la plus précise fournie par les unités de puissances trouvées dans les sous-opérateurs (c'est à dire celle qui contient le plus grand nombre de colonnes c_i). L'arbre d'additionneurs utilisé pour sommer q_0 et les $\Delta^i(x, c_i)$ décalés (c'est à dire multipliés par les q_i) peut lui aussi être partagé.

Dans l'unité fusionnée, quelques MSB de x pilotent les entrées de l'arbre d'additionneurs commun. Le reste des bits de x alimente l'unité commune de calcul des puissances de x. Lors de la fusion, le programme optimise le routage des entrées de l'arbre pour minimiser sa taille. Pour ceci, il essaie de faire correspondre au mieux les parties significatives (les $\Delta^i(x, c_i)$ décalés) des différents sous-opérateurs afin de minimiser la largeur des multiplexeurs et celle des additionneurs qu'ils alimentent.

L'étape de fusion décrite ci-dessus ne repose à aucun moment sur le fait que les sous-opérateurs approchent la même fonction. En conséquence, il est possible d'utiliser la même structure partagée pour calculer non seulement une fonction sur plusieurs sous-intervalles, mais aussi plusieurs fonctions sur plusieurs sous-intervalles. Le choix de la fonction à calculer est alors fait par une entrée supplémentaire de l'opérateur. Le programme implantant nos algorithmes est capable d'une telle fusion. Quelques exemples sont présentés en section 1.6.

1.5 Algorithme pour un opérateur évaluant une seule fonction

Dans cette section sont détaillées les étapes de la génération d'un opérateur évaluant une seule fonction sur un intervalle. On illustre le fonctionnement de l'algorithme par l'étude d'un opérateur pour la fonction sinus sur l'intervalle $[0, \pi/4[$, sans appliquer de transformation sur l'argument pour plus de simplicité. On pose $w_I = 12$ bits, et on vise une erreur moyenne d'environ $\mu = 2^{-12}$. Un polynôme de degré 3 est utilisé pour l'approximation, et la précision interne de l'opérateur est de 14 bits ($g = 2$).

L'approximation polynomiale de départ est estimée numériquement par la commande `minimax` de Maple. Le polynôme minimax de degré 3 pour la fonction sinus sur l'intervalle $[0, \pi/4[$, avec des coefficients arrondis à 10 décimales, est :

$$p^*(x) = -0.0000474553 + 1.0017332478\,x - 0.0095826177\,x^2 - 0.1522099691\,x^3.$$

Son erreur est d'au plus $\varepsilon_{th} = 4.75 \times 10^{-5}$, ce qui est équivalent à plus de 14.3 bits de précision, dans le cadre d'une évaluation exacte avec des coefficients d'une précision infinie pour l'évaluation du polynôme.

1.5.1 Étape 1 : quantification des coefficients p_i^*

Dans un premier temps, les multiplications utilisées pour l'évaluation des monômes $p_i^* \times x^i$ sont éliminées. Pour ce faire, la valeur théorique p_i^* est remplacée par une valeur q_i proche, mais qui peut être écrite sous la forme d'un nombre en borrow-save avec seulement un petit nombre n_i de chiffres non nuls. Ceci permet d'approcher la multiplication $p_i^* \times x^i$ à l'aide de $(n_i - 1)$ additions seulement. Une erreur proportionnelle à $|p_i^* - q_i|$ est faite, qui dépend à la fois de p_i^* et de n_i. Cette erreur peut être diminuée en accroissant la valeur de n_i. La figure 1.1 montre l'erreur commise lors de la quantification des valeurs écrites sur 12 bits fractionnaires qui sont comprises entre 0 et 1, pour $n_i = 3$ (en haut) et $n_i = 4$ (en bas).

Algorithme de quantification

L'algorithme de quantification d'un coefficient p_i^* en utilisant n_i bits non nuls avec l bits de précision est décrit dans l'algorithme 1. À chaque itération, la puissance de 2 la plus proche du reste partiel (initialisé à p_i^*) est calculée et ajoutée à q_i. On itère jusqu'à ce que n_i chiffres non nuls aient été ajoutés à q_i ou qu'une erreur inférieure à 2^{-l} ait été atteinte (la puissance de 2 la plus proche du reste est plus petite que 2^{-l}).

Si n_i est suffisant, le résultat de la quantification est une approximation de p_i^* écrite avec aussi peu de chiffres non nuls que possible et une erreur inférieure à 2^{-l}. S'il n'est pas possible d'approcher p_i^* avec une erreur inférieure à 2^{-l}, le coefficient quantifié est écrit sur n_i chiffres non nuls et aussi proche que possible de p_i^*.

On veut aussi que le coefficient retourné ait la plus petite largeur possible, c'est à dire la plus petite différence entre les rangs des bits de poids le plus fort et le plus faible. Par exemple, l'algorithme devrait retourner $(11.01)_{\mathsf{bs}}$ plutôt que $(10\overline{1}.01)_{\mathsf{bs}}$ pour la valeur 3.25 quantifiée avec $n_i = 3$ chiffres non nuls. À la première itération, on veut donc avoir $t = 1$ plutôt que $t = 2$. Pour ce faire, lorsqu'il reste plus d'un chiffre non nul à utiliser, au lieu d'incrémenter t lorsque $r > \frac{3}{2} \times 2^t$ on ne va l'incrémenter que si $r > \frac{13}{8} \times 2^t$. Le calcul de t devient :

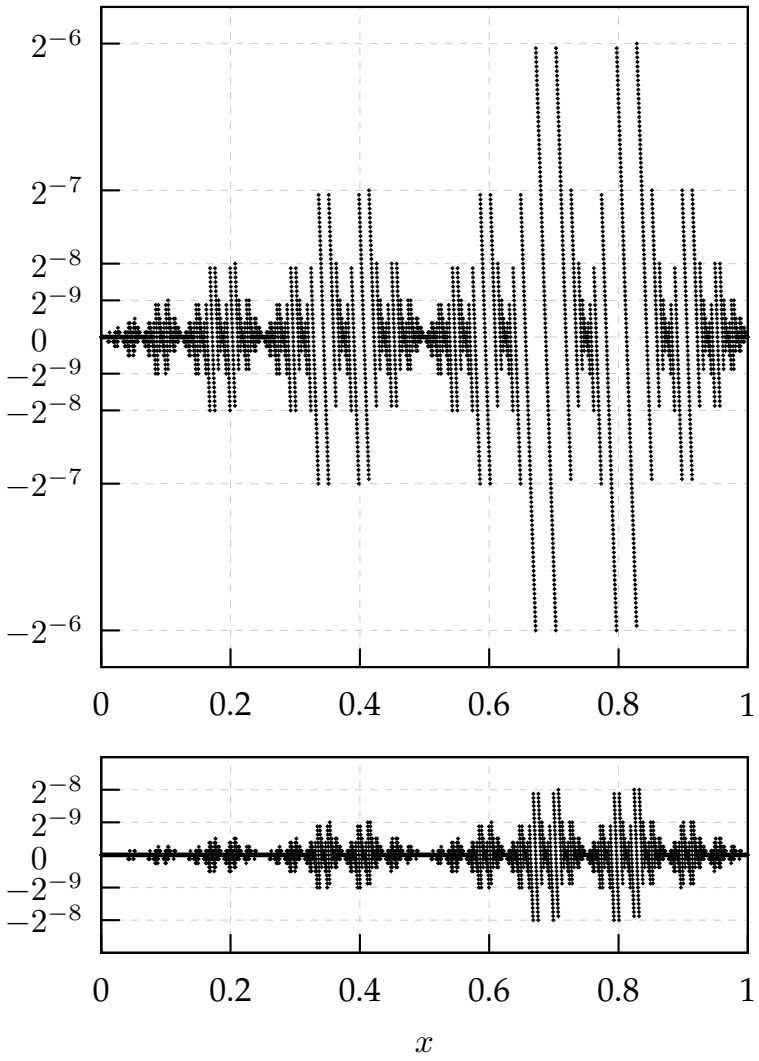

Figure 1.1 *Erreur de quantification des valeurs 12 bits sur l'intervalle* $[0, 1[$ *pour* $n_i = 3$ *(en haut) et* $n_i = 4$ *(en bas).*

$t \leftarrow \lfloor \log_2 |r| \rfloor$;
si $r > \frac{13}{8} \times 2^t$ **alors**
  $t \leftarrow t + 1$
sinon si $n + 1 \geq n_i$ ***et*** $r > \frac{3}{2} \times 2^t$ **alors**
  $t \leftarrow t + 1$

La condition $n + 1 \geq n_i$ permet de ne pas perdre de précision. En effet, sans cette condition, la quantification de 3.25 avec $n_i = 1$ chiffre non nul donnerait $(10)_{\text{bs}}$, ce qui est une approximation moins précise que $(100)_{\text{bs}}$.

Exemple 1.2 *La quantification de* $p_1^* = 1.0017332478$ *se fait avec différentes précisions suivant la valeur de* n_1 *:*

Si $\mathbf{n_1 = 1}$**,** $q_1 = 2^0 = (1.00000)_{bs}$ *approche* p_1^* *avec une précision de 9.1 bits (*$p_1^* - q_1 = 0.0017332478$*).*

Algorithme 1 : *Algorithme de quantification.*

Données : un coefficient p_i^*, une contrainte n_i et une précision l.
Résultat : q_i coefficient borrow-save proche de p_i^* écrit sur au plus n_i chiffres non nuls.

$n \leftarrow 0$; $q_i \leftarrow 0$; $r \leftarrow p_i^*$
$t \leftarrow \lfloor \log_2 |r| \rfloor$
si $r > \frac{3}{2} \times 2^t$ **alors**
  $t \leftarrow t + 1$
tant que $n < n_i$ ***et*** $t \geq -l$ **faire**
  $r \leftarrow r - 2^t \operatorname{signe}(r)$
  $q_i \leftarrow q_i + 2^t \operatorname{signe}(r)$
  $n \leftarrow n + 1$
  $t \leftarrow \lfloor \log_2 |r| \rfloor$
  si $r > \frac{3}{2} \times 2^t$ **alors**
    $t \leftarrow t + 1$

Si $\mathbf{n_1 = 2}$, $q_1 = 2^0 + 2^{-9} = (1.000000001)_{bs}$ *approche* p_1^* *avec une précision de 12.1 bits (*$p_1^* - q_1 = 0.0002198772$*).*

Si $\mathbf{n_1 = 3}$, $q_1 = 2^0 + 2^{-9} - 2^{-12} = (1.00000000100\overline{1})_{bs}$ *approche* p_1^* *avec une précision de 15.3 bits (*$p_1^* - q_1 = 0.0000242634$*).*

Selon le support d'implantation, certaines représentations peuvent être préférables à d'autres. Par exemple, dans une implantation sur ASIC, il peut s'avérer plus efficace de favoriser les représentations avec un minimum de chiffres négatifs, car une soustraction peut être plus coûteuse à réaliser qu'une addition. Dans ce cas, lors de la quantification de la valeur 1.375, la représentation $(1.011)_{\text{bs}}$ devra être choisie plutôt que la représentation apparemment équivalente $(1.10\overline{1})_{\text{bs}}$.

Quantification des coefficients d'un polynôme

On sait que le polynôme minimax est le meilleur polynôme d'approximation à coefficients réels d'une fonction pour la norme du maximum, en supposant que son évaluation est faite exactement. Cependant, dans un circuit des contraintes s'appliquent aux coefficients du polynôme et à son évaluation. Les coefficients sont par exemple arrondis vers une valeur en virgule fixe en précision finie. Il n'y a alors aucune garantie que le polynôme résultant de l'application de ces contraintes sur les coefficients du minimax soit bien la meilleure approximation polynomiale à coefficients contraints [14].

Afin d'améliorer la qualité de l'approximation $q(x)$ dont les coefficients (sauf q_0) sont quantifiés, on applique un procédé itératif où la différence entre la valeur du polynôme minimax p^* et celle du polynôme contraint q est réutilisée pour ajuster les q_i. L'algorithme part des monômes de faible degré, et réinjecte cette erreur dans le calcul des coefficients de degré supérieur (à ce niveau, le domaine d'étude est inclus dans $[-1, 1[$, donc les monômes ont d'autant moins d'impact que leur degré est élevé). Ce procédé itératif permet de corriger en partie l'erreur due à l'application des contraintes. On va ainsi obtenir une meilleure approximation contrainte de ce que donne l'application directe des contraintes aux coefficients p_i^* du minimax, comme c'était le cas dans l'article d'ASAP [3].

Calcul des contraintes de quantification n_i

Comme le montre la figure 1.1, les n_i ont une influence majeure sur la précision des coefficients quantifiés q_i par rapport aux coefficients réels p_i^* du minimax. Dans l'article d'ASAP, on fixait arbitrairement la valeur des n_i à 3. Il est désormais possible de déterminer automatiquement une valeur minimale pour les n_i qui donne un polynôme d'approximation approchant la fonction avec une précision suffisante. Le calcul des contraintes est décrit dans l'algorithme 2.

Algorithme 2 : *Calcul des n_i.*

Données : le polynôme minimax $p^*(x)$, l'erreur visée μ.
Résultat : les contraintes de quantification n_i permettant d'atteindre μ.
$(n_1, n_2, \ldots, n_d) \leftarrow (0, 0, \ldots, 0)$
On calcule $q(x)$ pour les contraintes $(n_1, n_2, \ldots, n_d)$.
tant que $\varepsilon_{avg}(q(x)) > \mu$ **faire**
  $e(x) \leftarrow$ `minimax` $((f-q)(y), y \in [a, b[, d)$
    $\leftarrow e_0 + e_1 x + \cdots + e_d x^d$
  pour $i = 1$ **à** d **faire**
    si $\int_{[a,b[} |e_i x^i| > \mu$ **alors**
      $n_i \leftarrow n_i + 1$
  On recalcule les coefficients quantifiés de $q(x)$ avec les nouvelles contraintes $(n_1, n_2, \ldots, n_d)$.

Le polynôme $e(x)$ qui apparaît dans l'algorithme sert à caractériser l'erreur commise en évaluant la fonction f à l'aide d'un polynôme quantifié q. Chaque monôme $e_i x^i$ de $e(x)$ peut être vu comme la contribution à l'erreur du monôme de degré i, due à la fois à l'erreur du minimax et à la quantification du coefficient q_i.

Exemple 1.3 *On calcule des valeurs des contraintes n_i pour une approximation polynomiale de degré 3 de la fonction sinus, avec une précision visée de 12 bits. À chaque étape, la contrainte n_i en gras a été incrémentée par l'algorithme, parce que le monôme de degré i de $e(x)$ indiquait une trop grande contribution à l'erreur.*

(n_1, n_2, n_3)	ν_{avg} *(en bits)*	ν_{max} *(en bits)*
$(0, 0, 0)$	1.1	0.2
$(\mathbf{1}, \mathbf{1}, \mathbf{1})$	7.3	5.3
$(\mathbf{2}, \mathbf{2}, \mathbf{2})$	9.8	7.5
$(2, 2, \mathbf{3})$	12.6	9.2

Les contraintes n_i calculées par l'algorithme reflètent le besoin en précision de chaque coefficient quantifié q_i. Elles dépendent de la magnitude de p_i^*, du degré i du monôme (on a $x \in [-1, 1[$, donc un monôme a en général d'autant moins d'influence que son degré est haut), et de la difficulté d'approcher p_i^* par un coefficient quantifié (figure 1.1). Bien que le coefficient $p_1^* = 1.0017332478$ ait une forte amplitude, $n_1 = 2$ suffit car p_1^* est proche (à 12.1 bits près) de $q_1 = 2^0 + 2^{-9}$. Par contre, $p_3^* = -0.1522095004$ est plus petit, mais $n_3 = 3$ chiffres non nuls sont requis pour avoir une approximation suffisamment proche $q_3 = -2^{-3} - 2^{-5} + 2^{-8}$ (à 12.9 bits près).

1.5.2 Étape 2 : approximation des puissances x^i

Une fois les coefficients du polynôme quantifiés, on a toujours besoin de quelques multiplieurs pour évaluer $q(x) = q_0+q_1x+q_2x^2+\ldots+q_dx^d$. En effet, il faut calculer les puissances x^i de l'argument. Puisqu'un schéma d'évaluation parallèle est utilisé, on a besoin de toutes ces puissances en même temps. Deux manières de les calculer sont présentées et comparées ici. La première repose sur des opérateurs optimisés calculant chacun une puissance. La seconde utilise un réseau de multiplieurs de Booth modifiés et de squarers, qui implantent un équivalent matériel de l'algorithme d'exponentiation rapide.

Opérateurs spécialisés pour le calcul des puissances

Dans [3], les estimations des puissances de x étaient obtenues en développant le tableau de produits partiels (Partial Products Array) de x^i puis en y appliquant des simplifications logiques classiques utilisées pour le carré [11, p.221], [13, p.201]. Ces simplifications reposent sur l'application de règles logiques qui permettent de diminuer le nombre de produits partiels ou de modifier leur répartition.

Dans le cadre de l'extension de la méthode présentée à ASAP [3], nous avons étudié les opérateurs pour les puissances entières, et obtenu de nouvelles réductions logiques ainsi qu'une stratégie de réduction efficace. Ces travaux ont été présentés [4] lors de la conférence SPIE en 2006. Les nouveaux opérateurs pour le calcul des puissances sont obtenus par une méthode hybride basée sur ces travaux et ceux présentés dans [31]. Une description détaillée en est faite en section 1.9.

Structure *square-and-multiply* pour le calcul des puissances

Notre programme est aussi capable de calculer les puissances de x à l'aide d'un équivalent matériel de l'algorithme d'*exponentiation rapide* (*square-and-multiply*). Cet algorithme calcule la puissance $i$ème de x récursivement de la façon suivante :

$$x^i = \begin{cases} x, & \text{si } i = 1, \\ (x^2)^{i/2} & \text{si } i \text{ est pair,} \\ x \times (x^2)^{(i-1)/2} & \text{si } i > 2 \text{ est impair.} \end{cases}$$

Par exemple, x^5 est calculé sous la forme $x \times ((x^2)^2)$. Dans le circuit, les composants sont partagés entre les différentes puissances, c'est à dire que x^2 sert à calculer x^4 qui lui-même est utilisé pour obtenir x^5. La figure 1.2 montre une unité de ce type utilisée pour le calcul exact des puissances 2 à 6 de x.

Estimation des puissances

Les opérateurs calculant les puissances décrits ci-dessus retournent un résultat exact, ce qui est inutile dans le cadre de notre application. Il est donc possible de réduire la surface du circuit en appliquant un schéma de troncature aux tableaux de produits partiels qu'ils contiennent. Dans l'article d'ASAP [3], on appliquait un schéma de troncature directe, qui consiste à n'implanter qu'une partie du tableau de produits partiels correspondant aux colonnes de poids fort.

Le schéma de troncature variable [30] (VCT) est un bon candidat. Il s'agit d'un schéma dépendant des données, qui donne une précision meilleure que le schéma de troncature directe. On aura donc besoin de moins de colonnes dans les tableaux de produits partiels pour atteindre la même

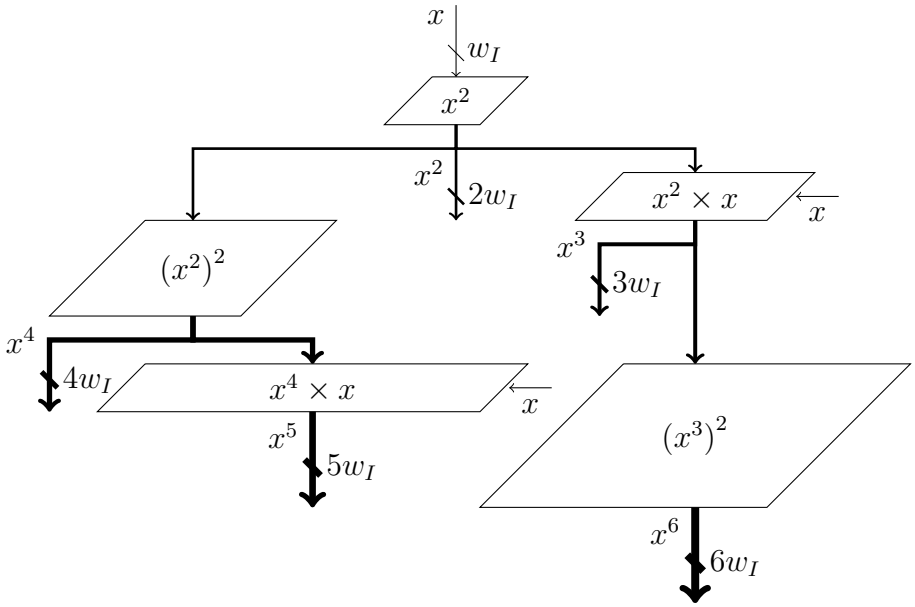

Figure 1.2 *Structure square-and-multiply pour le calcul exact des puissances 2 à 6 de x.*

précision, d'où des opérateurs plus petits. Le schéma de troncature VCT approche la valeur des retenues générées dans la partie tronquée en utilisant la première colonne des produits partiels tronqués, qui est multipliée par 2 (décalée d'une position vers la gauche). Les produits partiels de poids plus faibles sont omis.

Lorsqu'on applique ce schéma aux opérateurs spécialisés pour le calcul des puissances, on déplace des produits partiels dans le PPA : il est possible d'appliquer de nouveau le procédé de réduction décrit en section 1.9 pour diminuer encore plus la surface de l'unité.

L'estimation tronquée de x^i, notée $\Delta^i(x, c_i)$, est caractérisée par le nombre de colonnes c_i du tableau de produits partiels qui sont conservées et implantées. Ce tableau tronqué est soit celui d'un opérateur optimisé, ou vient des multiplieurs et unités d'élévation au carré d'une structure square-and-multiply.

Calcul des contraintes d'estimation c_i

La valeur minimale de c_i qui permet d'obtenir une estimation suffisamment précise de x^i dépend principalement de p_i^*. En effet, plus p_i^* est petit et plus on peut se permettre d'utiliser une estimation grossière de x^i dans l'approximation du monôme $p_i^* x^i$. Les contraintes c_i étaient fixées arbitrairement dans l'article d'ASAP [3]. Une procédure automatique est maintenant intégrée à l'algorithme qui permet d'obtenir les valeurs minimales suffisantes des c_i pour les deux types d'unités de calcul des puissances.

Dans le cas des puissances calculées en parallèle dans des opérateurs spécialisés, le nombre de colonnes c_i utilisées pour l'estimation de x^i est calculé en partant de $c_i = 1$ et incrémentant c_i jusqu'à ce que l'erreur faite en approchant le monôme $p_i^* x^i$ par $q_i \times \Delta^i(x, c_i)$ soit inférieure à μ.

Pour une structure square-and-multiply, l'estimation de x^i est réutilisée dans le calcul des puissances suivantes, comme le montre la figure 1.2. Il faut donc non seulement que les c_i soient suffisants pour que $q_i \times \Delta^i(x, c_i)$ approche $p_i^* x^i$ précisément, mais aussi qu'en réutilisant $\Delta^i(x, c_i)$ on soit capable d'approcher les monômes suivants précisément. Pour ce faire, on détermine les c_i

dans l'ordre croissant. On part de $c_i = 1$ et on l'incrémente jusqu'à ce que les approximations de $p_i^* x^i$ et de tous les $p_j^* x^j$ avec $j > i$ (en supposant que les multiplications et carrés aux étapes $j > i$ sont calculés exactement) soient toutes assez précises.

Le coût matériel du calcul exact des puissances est proportionnel à $(w_I)^d$ pour les opérateurs optimisés parallèles, et proportionnel à $(w_I)^2$ seulement pour la structure square-and-multiply. Les optimisations, en particulier le tri et les réductions logiques, ne servent qu'à diminuer la constante de proportionnalité. On pourrait donc penser que les puissances parallèles ne sont pas aussi efficaces qu'une structure square-and-multiply. C'est en effet le cas pour un calcul exact des puissances, comme on peut le voir dans [31] pour un calcul du cube. L'opérateur spécialisé devient plus gros pour $w_I \geq 16$.

Cependant, on ne calcule ici qu'une estimation des puissances, afin de remplir un critère de précision finale. Les opérateurs tronqués, pour des degrés allant jusqu'à 6 et des précisions d'au plus 24 bits, demandent en général des ressources similaires pour les deux manières de calculer les puissances. Pour des fonctions impaires comme par exemple le sinus, les puissances parallèles sont souvent un meilleur choix, car les puissances de x y sont calculées indépendamment. Les puissances paires sont alors évaluées très grossièrement. Dans une structure de type square-and-multiply, on est obligés d'évaluer les puissances paires (par exemple x^2) avec une précision suffisante pour que les puissances impaires qui en dépendent ($x^3 = x \times (x^2)$) soient assez précises.

Exemple 1.4 *On calcule les c_i pour une évaluation sur 12 bits de la fonction sinus, avec un polynôme de degré 3.*
Si on calcule les puissances de x à l'aide d'opérateurs spécialisés parallèles (à gauche), on incrémente c_i jusqu'à ce que la précision moyenne (notée ν_i) obtenue en approchant le monôme $p_i^* x^i$ par $q_i \times \Delta^i(x, c_i)$ atteigne 12 bits.
Dans le cas d'une structure square-and-multiply (à droite), c_i doit aussi être suffisant pour que la précision des monômes de degré $j > i$ soit d'au moins 12 bits.

Opérateurs spécialisés		*Unité square-and-multiply*			
	ν_2			ν_2	ν_3
$c_2 =$ 1	10	$c_2 =$ 1		10	7.25
2	10.75		2	10.25	7.5
3	11.5		3	11.5	8.75
4	12.25		4	12	9.5
			⋮	⋮	⋮
			7	14.5	12.5
$c_3 =$ 1	6	$c_3 =$ 1			6
⋮	⋮		⋮		⋮
8	11.5		8		11.75
9	12.5		9		12.25

Les c_i reflètent les besoins en précision pour chaque estimation de la valeur x^i.
Dans le cas des puissances parallèles, $p_2^* \simeq -0.01$ est très petit, donc $c_2 = 4$ colonnes donne assez de précision pour une estimation $q_2 \times \Delta^2(x, c_2)$ de $p_2^* x^2$. Par contre $p_3^* \simeq -0.15$ est plus grand, il faut au moins $c_3 = 9$ colonnes pour obtenir une estimation précise à 12 bits du monôme de degré 3.

Dans le cas d'une unité square-and-multiply, ceci reste vrai, mais l'estimation qu'on fait de x^3 dépend de celle de x^2 (car $\Delta^2(x, c_2)$ est utilisé pour calculer $\Delta^3(x, c_3)$). Il faut donc au moins $c_2 = 7$ colonnes dans l'estimation de x^2.

1.5.3 Étape 3 : ajustement des coefficients

Il n'y a plus de multiplications exactes dans l'évaluation de la fonction. L'opérateur calcule maintenant l'approximation :

$$\tilde{q}(x) = q_0 + q_1 \times \Delta^1(x, c_1) + \ldots + q_d \times \Delta^d(x, c_d).$$

On sait qu'il se peut que l'approximation contrainte obtenue, qui découle de l'approximation minimax, ne soit pas optimale. En particulier, ses coefficients peuvent souvent être légèrement modifiés pour améliorer la qualité numérique de l'estimation $\tilde{q}$ de f. L'erreur en sortie de l'opérateur est notée $\tilde{\varepsilon}(x) = f(x) - \tilde{q}(x)$.

Ajustement du terme additif q_0

Le terme additif q_0 de l'approximation polynomiale n'est pas quantifié. Il est habituellement possible de l'ajuster pour diminuer l'erreur absolue moyenne de l'évaluation. Le but de l'ajustement sur q_0 est de minimiser $\varepsilon_{\mathsf{avg}}$, c'est à dire déterminer la valeur δ qui minimise la somme des $|\tilde{\varepsilon}(x) - \delta|$ pour tous les x dans le domaine $[a, b[$. La valeur δ est donc *une* médiane de l'ensemble des erreurs $\tilde{\varepsilon}(x)$ (lorsqu'il y a un nombre pair n d'éléments dans l'ensemble des erreurs, on prend comme médiane la $n/2$ème valeur, et non la moyenne des $n/2$ et $(n/2+1)$ème valeurs). Son calcul est détaillé dans l'algorithme 3.

Algorithme 3 : *Ajustement de q_0.*

Données : la fonction f, et le polynôme d'approximation $\tilde{q}(x)$.
Résultat : La valeur de q_0 pour laquelle l'erreur absolue moyenne est minimale.
Soit t un tableau
$i \leftarrow 0$
pour chaque x **dans** $[a, b[$ **faire**
  $t[i] \leftarrow \tilde{\varepsilon}(x)$
  $\leftarrow (f - \tilde{q})(x)$
  $i \leftarrow i + 1$
Trier le tableau t
$q_0 \leftarrow q_0 - t[\frac{i}{2}]$

Exemple 1.5 *En utilisant une structure square-and-multiply, la précision moyenne de notre exemple avant ajustement est d'environ 11.75 bits. L'algorithme modifie $q_0 = -2^{-14}$ en $q_0 = +2^{-13}$.*
La précision moyenne ν_{avg} est alors portée à 12.25 bits.

Ajustement des coefficients q_i

Dans la mesure où les coefficients du polynôme p^* et les puissances de x sont seulement approchés, il arrive qu'une valeur q_i puisse être ajustée pour que $q_i \times \Delta^i(x, c_i)$ fournisse une meilleure approximation de $p_i^* \times x^i$.

L'algorithme utilisé est une généralisation du procédé d'ajustement pour q_0. On veut cette fois rendre aussi petite que possible l'erreur absolue moyenne sur un monôme, c'est à dire déterminer la valeur δ qui minimise la somme des $|p_i^* x^i - (q_i - \delta)\Delta^i(x, c_i)|$ pour tous les $x \in [a, b[$. La valeur δ est maintenant une médiane de l'ensemble des erreurs relatives $\varepsilon_i(x) = p_i^* x^i / \Delta^i(x, c_i) - q_i$, pondérées par $|\Delta^i(x, c_i)|$. La valeur δ est obtenue en triant l'ensemble de ces valeurs dans l'ordre lexicographique (erreur, poids), puis en prenant l'erreur du couple qui partitionne l'ensemble en deux sous-ensembles de même poids totaux. L'algorithme 4 décrit l'ajustement des q_i.

Algorithme 4 : *Ajustement des q_i.*

Données : le monôme théorique $p_i^* x^i$ et celui d'approximation $q_i \times \Delta^i(x, c_i)$.
Résultat : La valeur de q_i pour laquelle l'erreur absolue moyenne est minimale sur le monôme.

```
Soit t un tableau de couples (erreur, poids)
j ← 0 ; s ← 0
pour chaque x dans [a, b[ faire
    si Δ^i(x, c_i) ≠ 0 alors
        ε_i(x) = p_i^* x^i / Δ^i(x, c_i) − q_i
        t[j] ← (ε_i(x), |Δ^i(x, c_i)|)
        s ← s + |Δ^i(x, c_i)|
        j ← j + 1
Trier t dans l'ordre lexicographique
j ← 0
tant que s > 0 et |s| > 2 × poids(t[j]) faire
    s ← s − 2 × poids(t[j])
    j ← j + 1
q_i ← quantifier(q_i − erreur(t[j]), n_i)
```

Commentaire

L'étape d'ajustement était une fonction heuristique dans l'article d'ASAP [3]. Elle est désormais effectuée par un algorithme déterministe qui retourne une valeur optimale. On remarque que les deux fonctions décrites ne font aucune hypothèse sur les valeurs initiales de q_0 et q_i. Il est donc possible de s'en servir pour calculer ces valeurs et pas seulement les ajuster. L'ajustement de q_0 est utilisé de façon importante dans l'étape finale d'exploration décrite en section 1.5.4, où il sert à calculer q_0.

Il existe des algorithmes plus efficaces que ceux présentés ici pour calculer la médiane sans faire appel à un tri [23, 25]. Ceux-ci seront prochainement implantés, afin d'accélérer l'ajustement de q_0, dont dépend l'étape d'exploration.

1.5.4 Étape 4 : exploration

Motivation

Une fois connues les valeurs des n_i, q_i et c_i, les procédures d'ajustement sont utilisées pour améliorer la qualité numérique de l'approximation $\tilde{q}(x)$. Cependant, il arrive que l'approximation résultante ne soit toujours pas optimale, bien que proche de l'optimal.

Description de la recherche

Pour y remédier, on peut effectuer une dernière étape optionnelle. Elle consiste en une exploration sur les coefficients quantifiés voisins des q_i. Cette exploration peut être paramétrée par la largeur du voisinage des q_i à explorer, c'est à dire le nombre de coefficients quantifiés précédents ou suivant q_i (à n_i donné) que l'on teste. On peut aussi paramétrer cette recherche par le temps de calcul disponible. Il est en général difficile d'estimer le temps d'exploration en fonction du voisinage. Le simple calcul des coefficients quantifiés précédent et suivant q_i en fonction de n_i se révèle difficile.

Cette exploration nous fournit habituellement de nouvelles solutions un peu plus précises. De plus, elle va aussi découvrir de nouvelles solutions avec une erreur moyenne très proche de celle fournie par la meilleure solution, mais avec des valeurs n_i moindres. Ces solutions généreront donc des opérateurs plus petits.

> **Exemple 1.6** *Considérons l'étude de la fonction sinus, avec des opérateurs de puissances spécialisés. La meilleure solution que l'on trouve avant l'exploration offre une précision moyenne de 12.1 bits justes avec* $n_1 = 2$, $n_2 = 2$ *et* $n_3 = 3$. *Une exploration sur un voisinage de largeur 10 autour des coefficients quantifiés prend une vingtaine de secondes sur ordinateur. Elle permet d'obtenir une nouvelle solution d'une précision moyenne de 12.1 bits, mais pour seulement* $(n_1, n_2, n_3) = (1, 1, 2)$ *chiffres non nuls. Elle donne aussi une solution avec* $(n_1, n_2, n_3) = (1, 1, 3)$ *qui offre une précision moyenne de 12.6 bits.*

L'ajustement du terme additif q_0 devient crucial à cette étape. Par exemple, une des solutions obtenues pendant l'exploration est une évaluation d'une précision moyenne de 11.3 bits seulement pour $q_0 = +2^{-13}$ (la valeur de q_0 avant l'exploration). Après ajustement, $q_0 = -2^{-12}$ donne une nouvelle précision moyenne de 12.25 bits.

Les opérateurs présentés dans ce chapitre sont obtenus par des explorations durant au plus une douzaine d'heures. Ces explorations peuvent être fortement accélérées en évaluant dans un premier temps les solutions sur un échantillon de valeurs. Pour une précision de 24 bits, on va d'abord tester une dizaine de milliers d'entrées aléatoires (au lieu des 2^{24} entrées possibles) et en déduire une estimation de ε_{avg}. Si on est proche de la meilleure solution, on recalcule ε_{avg} exactement, sinon on passe à la solution suivante.

On gagne ainsi plusieurs ordres de grandeurs dans le temps d'exploration. Lors de comparaisons, nous n'avons pas vu se perdre de solutions pour des tailles de l'échantillon suffisantes (en pratique environ 5000 entrées).

1.5.5 Commentaire sur l'ajustement et l'exploration

Lors de l'étude la fonction sinus avec une précision de 12 bits, évaluée à l'aide d'un polynôme de degré 3, plusieurs polynômes sont apparus. On utilise comme point de départ le polynôme minimax :

$$p^*(x) = -0.0000474553 + 1.0017332478\,x - 0.0095826177\,x^2 - 0.1522099691\,x^3$$

Après détermination des contraintes de quantification $(n_1, n_2, n_3) = (2, 2, 3)$, on arrive à un polynôme dont les coefficients sont quantifiés :

$$q_0(x) = -2^{-14} + (2^0 + 2^{-9})\,x + (-2^{-7} - 2^{-9})\,x^2 + (-2^{-3} - 2^{-5} + 2^{-8})\,x^3$$

On détermine ensuite, pour une unité qui estime les puissances de x dans des opérateurs spécialisés parallèles, que le nombre de colonnes minimal pour l'estimation de x^2 est 4, et celui pour l'estimation de x^3 est 9.

Après les étapes d'ajustement et l'exploration, deux approximations $\tilde{q_1}$ et $\tilde{q_2}$ sont trouvées, qui génèrent des opérateurs plus précis (12.1 bits et 12.625 bits de précision moyenne respectivement) et plus petits ($(1, 1, 2)$ et $(1, 1, 3)$ bits non nuls dans les coefficients respectivement) que q_0, qui offre 12 bits de précision moyenne pour $(n_1, n_2, n_3) = (2, 2, 3)$ bits non nuls.

$$q_1(x) = -2^{-13} + 2^0\,x - 2^{-8}\,x^2 + (-2^{-3} - 2^{-5})\,x^3$$

$$q_2(x) = 0 + 2^0\,x - 2^{-8}\,x^2 + (-2^{-3} - 2^{-5} + 2^{-10})\,x^3$$

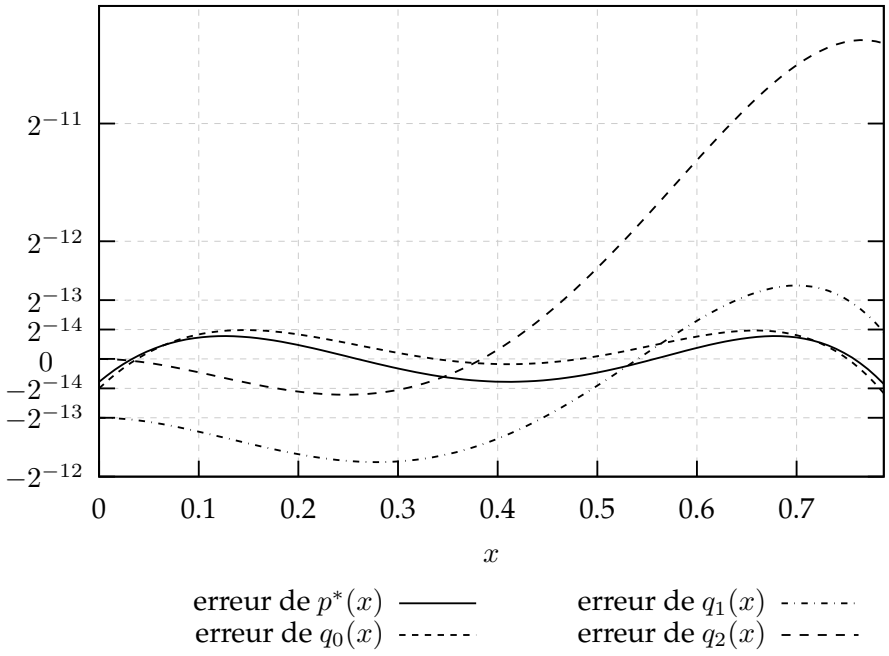

Figure 1.3 *Erreur du polynôme minimax $p^*(x)$, du polynôme quantifié $q_0(x)$, et des deux approximations $q_1(x)$ et $q_2(x)$. Les puissances et les calculs intermédiaires sont exacts.*

La figure 1.3 montre que, dans le cadre d'une évaluation polynomiale exacte, q_1 et q_2 sont d'une qualité numérique bien moindre à celle de q_0. Il n'y a donc aucune raison de les considérer à priori.

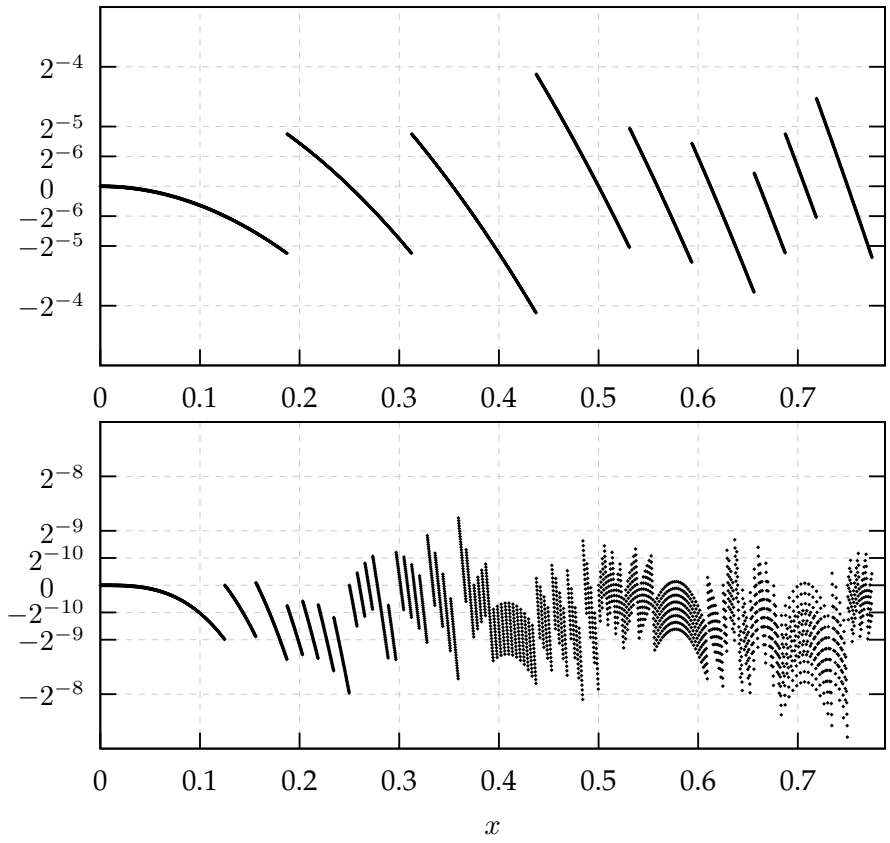

Figure 1.4 *Erreur commise lors de l'estimation du carré sur 4 colonnes (en haut) et du cube sur 9 colonnes (en bas).*

C'est l'estimation des puissances qui les rend intéressants. En utilisant $\Delta^2(x, 4)$ et $\Delta^3(x, 9)$ à la place des valeurs exactes de x^2 et x^3, on introduit une erreur qu'il est difficile de caractériser (voir figure 1.4, en haut pour x^2 et en bas pour x^3).

Cette erreur s'ajoute à celle introduite par la quantification des coefficients, et les erreurs d'arrondi qui sont dues aux calculs intermédiaires sur un nombre fini de bits. L'erreur d'évaluation par rapport à la fonction sinus devient complexe.

La figure 1.5 montre l'erreur d'évaluation pour $\tilde{q_0}(x)$ (en haut) et $\tilde{q_2}(x)$ (en bas). On constate que q_2, qui dans le cadre d'une évaluation exacte est un mauvais candidat, offre une bien meilleure précision pour une évaluation contrainte. Il n'est pas possible de le savoir à priori, c'est pourquoi l'étape d'exploration, et les procédures d'ajustement qu'elle utilise, sont nécessaires.

1.6 Résultats

1.6.1 Simulations VHDL

Les descriptions VHDL des opérateurs ont été testées par simulation sur Modelsim pour s'assurer que les opérateurs générés se comportent de manière identique au modèle logiciel utilisé par le programme. Les courbes d'erreur obtenues sont identiques à celles présentées dans les exemples de la section précédente.

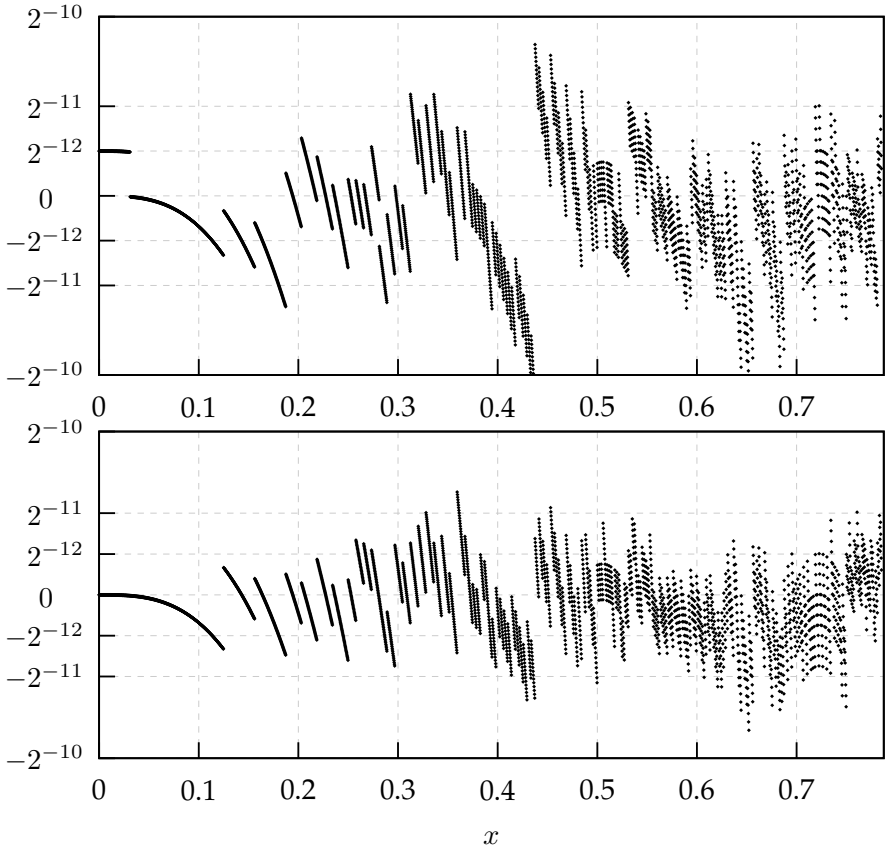

Figure 1.5 *Erreur d'évaluation de la fonction sinus pour les approximations $\tilde{q_0}(x)$ (en haut) et $\tilde{q_2}(x)$ (en bas).*

1.6.2 Implantation sur FPGA

Différents opérateurs ont été générés par notre programme pour plusieurs fonctions, degrés d'approximation et découpages des domaines. Les résultats de synthèse sont fournis par la suite logicielle ISE 8.2 pour un FPGA Virtex II v1000. La synthèse est faite avec un effort de synthèse orienté en surface, et le placement-routage est effectué avec un effort d'optimisation important. L'utilisation de scripts et d'outils de génération automatiques nous a permis d'obtenir de nombreux résultats d'implantation. Une partie des résultats d'implantation est reportée en table 1.1.

1.6.3 Opérateurs pour des fonctions multiples

La table 1.2 montre qu'un unique opérateur pour l'évaluation des fonctions $\sin(x)$, $2^x - 1$ et $\log(1 + x)$ est jusqu'à 44% plus petit que la somme des surfaces des opérateurs calculant ces fonctions séparément.

Ce résultat s'explique facilement par le partage de l'unité calculant les puissances et de l'arbre d'addition. Le seul surcoût est celui du routage et des multiplexeurs utilisés pour la sélection de la fonction et de l'intervalle d'évaluation. Le gain de matériel devient d'autant plus important que le nombre de fonctions et d'intervalles est grand.

Fonction	w_I (en bits)	8	12	16	20	24
$\sin(x)$	taille (tranches)	33	76	184	355	647
	délai (ns)	11.6	18.8	25.7	39	40.3
	ν_{avg} (bits)	8.6	12.6	16.7	20.5	24.4
$\log(x+1)$	taille (tranches)	42	86	175	395	599
	délai (ns)	13.8	18.3	23.3	36	39.1
	ν_{avg} (bits)	8.6	12.6	16.8	20.5	24.6
$2^x - 1$	taille (tranches)	30	85	171	356	736
	délai (ns)	11.9	20.5	22.7	37.5	43.1
	ν_{avg} (bits)	8.2	12.8	16.6	20.1	24.2

Table 1.1 *Résultats d'implantation obtenus par notre méthode. ($\nu_{avg} = -\log_2(\varepsilon_{avg})$)*

w_I (en bits)	8	12	16	20	24
Implantations séparées	105	247	530	1106	1982
Opérateur fusionné	65	153	312	633	1095
Gain en surface	38%	38%	41%	42%	44%

Table 1.2 *Tailles obtenues (en tranches) pour des implantations séparées et fusionnées des fonctions $\sin(x)$ sur le domaine $[0, \frac{\pi}{4}[$, et $2^x - 1$ et $\log(1+x)$ sur $[0, 1[$.*

1.7 Comparaisons avec des méthodes existantes

Il n'existe pas à notre connaissance d'autres méthodes dont le but est d'évaluer une fonction avec une erreur absolue moyenne minimale, éventuellement au détriment de l'erreur maximum. Les méthodes existantes minimisent l'erreur maximum, habituellement à moins d'un LSB(arrondi fidèle) ou moins d'un demi LSB(arrondi correct). La méthode proposée a donc été comparée à quelques méthodes existantes pour l'arrondi fidèle. La méthode SMSO [26] évalue la fonction $\sin(\frac{\pi}{4}x)$ sur l'intervalle $[0, 1[$ au lieu de $\sin(x)$ sur $[0, \frac{\pi}{4}[$. Cette fonction permet une meilleure utilisation du calcul des puissances, et est plus « étalée », donc plus facile à approcher. Nous avons donc étudié la même fonction dans les comparaisons. Les opérateurs multipartites évaluent quant à eux $\sin(x)$ sur $[0, \frac{\pi}{4}[$, mais la différence en termes de surface avec notre méthode reste très importante si notre méthode évalue aussi $\sin(x)$.

La figure 1.6 illustre, pour ces trois méthodes, la surface des opérateurs en fonction de la précision dans le cas de la fonction sinus. Il est à noter que notre méthode assure une précision moyenne proche de la précision de l'argument (c'est pourquoi les croix ne sont pas exactement en face des graduations), alors que les tables multipartites et SMSO fournissent un arrondi fidèle (une erreur inférieure au LSBde l'argument).

Notre méthode génère des opérateurs 15% plus petits que SMSO pour une précision de 24 bits. De plus, les méthodes multipartites et SMSO sont dédiées à une unique fonction. La taille d'un opérateur est donc linéaire en le nombre de fonctions qu'il évalue. Par contre notre méthode permet le partage de matériel entre plusieurs évaluations de fonctions, et devient d'autant plus intéressante que le nombre de fonctions à évaluer est grand.

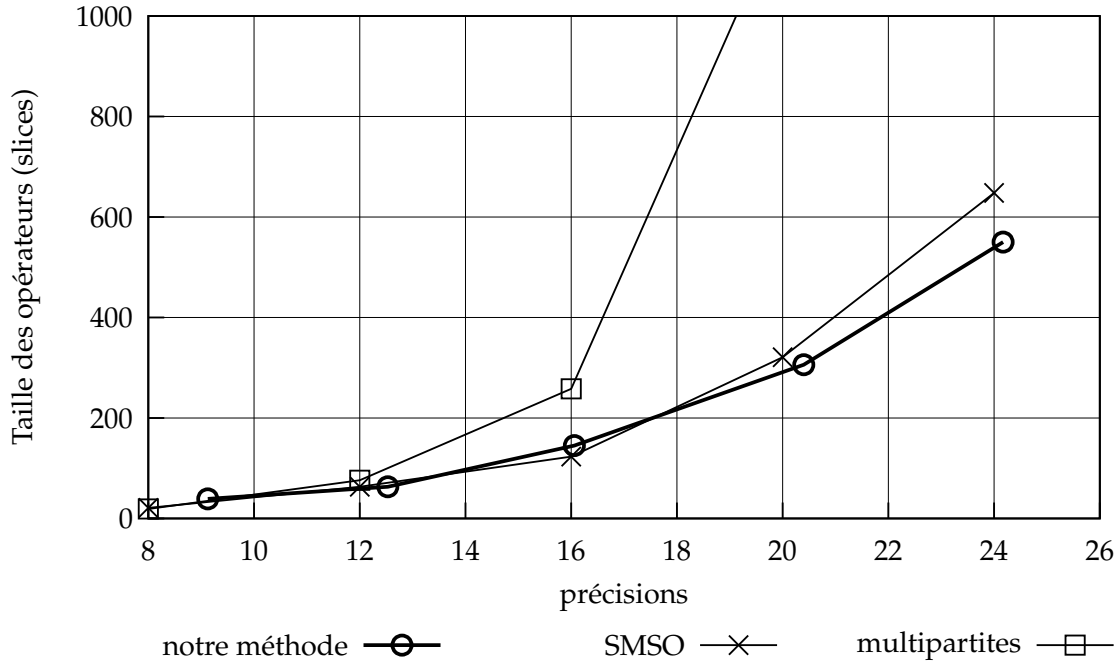

Figure 1.6 *Comparaison des tailles d'opérateurs utilisant plusieurs méthodes pour la fonction sinus*

1.8 Conclusion

Nous avons présenté un algorithme pour la génération automatique d'opérateurs matériels évaluant des fonctions multiples. Il repose sur l'utilisation d'approximations polynomiales avec des coefficients creux et des puissances tronquées. Les multiplications exactes dans l'évaluation du polynôme sont remplacées par de petits opérateurs spécialisés ou de courtes séquences d'additions. Une grande partie du matériel est partagée entre l'évaluation des différentes fonctions.

Ce chapitre est une extension du travail présenté dans [3], où les opérateurs générés étaient limités à une précision de 16 bits et évaluaient une seule fonction. Il est désormais possible d'évaluer plusieurs fonctions au sein d'un même opérateur. Les paramètres de génération sont tous automatiquement calculés de façon efficace. Un nouveau schéma de troncature plus précis est utilisé pour l'estimation des puissances. Ces améliorations nous permettent d'atteindre une précision de 24 bits et des opérateurs plus petits, qui conviennent à des applications en multimédia et en traitement du signal

Dans l'avenir, nous comptons étendre le générateur pour fournir du VHDL destiné à des cibles de type ASIC, et mettre en pratique nos opérateurs dans des applications de traitement du signal.

1.9 Opérateurs spécialisés pour le calcul des puissances

Cette section détaille la façon dont ont été obtenus les opérateurs spécialisés pour le calcul des puissances entières de x. Ces unités sont basées sur deux méthodes. La première est une extension des travaux de Liddicoat et Flynn sur le calcul du cube [31]. La seconde méthode est une étude d'opérateurs calculant des puissances entières que nous avons présenté récemment [4].

Dans ce travail nous présentons des améliorations d'opérateurs matériels dédiés au calcul de puissances avec un exposant entier fixé (x^3, x^4, ...) en virgule fixe binaire ou en représentation entière. La méthode proposée réduit le nombre de produits partiels en utilisant des simplifications basées sur de nouvelles identités et transformations. Des simplifications sont faites à la fois aux niveaux logiques et arithmétiques. La méthode proposée a été implantée sous la forme d'un générateur VHDL qui produit des descriptions synthétisables d'opérateurs optimisés. Les résultats ont été illustrés sur des circuits FPGA.

1.9.1 Motivation

En dehors de l'utilisation faite dans ce chapitre, le carré, le cube et l'élévation à des puissances supérieures avec un exposant entier fixe sont souvent utilisés en traitement du signal ou dans des applications graphiques [35, 37]. Dans des implantations sur circuits numériques de ces opérateurs d'exponentiation, les multiplieurs sont souvent remplacés par des unités spécialisées pour le calcul des puissances. Ces opérateurs utilisent des identités logiques pour réduire le tableau de produits partiels, ou PPA (pour Partial Products Array). Grâce à des symétries et d'autres types de simplifications possibles dans le PPA, les opérateurs spécialisés sont plus petits et plus rapides qu'une implantation à base de multiplieurs.

Les opérateurs proposés ici utilisent à la fois des simplifications aux niveau logiques et arithmétiques. Les simplifications classiques sont étendues et généralisées. On introduit aussi de nouvelles identités qui améliorent la réduction du PPA. L'ordre d'application de ces identités est un problème majeur si on veut minimiser la surface du circuit. Ce sujet n'est pourtant pas couvert dans la littérature. On propose ici une méthode qui permet une réduction rapide du tableau de produits partiels. Notre méthode est applicable pour x^n, où n est un entier fixé supérieur ou égal à 3 (Pour $n = 2$, notre méthode n'apporte pas d'amélioration par rapport aux méthodes existantes).

Cette section est organisée de la façon suivante : quelques notations utiles sont introduites en section 1.9.2, puis un état de l'art du domaine est dressé en 1.9.3. Les améliorations proposées sont décrites dans la section 1.9.4, suivies de quelques résultats d'implantation en section 1.9.6. Enfin, nous présentons la combinaison entre notre méthode et l'extension des travaux présentés dans [31] qui a été utilisée pour le calcul des puissances dans les opérateurs spécialisés utilisés dans ce chapitre.

1.9.2 Notations

Cette section traite du calcul de x^n. L'argument x est une valeur de w bits en virgule fixe binaire non signée, ou dans un format entier. Les bits de x sont notés x_{w-1}, x_{w-2}, ..., x_0. L'exposant n est un entier fixé, vérifiant $n \geq 3$.

Les formules logiques se lisent de la façon suivante :

– $\overline{a}$ signifie « non a ».

– $a \wedge b$, parfois noté ab, se lit « a et b ».

– $a \vee b$ veut dire « a ou b ».
– $a \Rightarrow b$ signifie « a implique b »,

Le tableau de produits partiels (PPA) est représenté graphiquement comme sur la figure 1.7 (cas du cube d'un entier non signé sur 4 bits). Tous les produits partiels (PP) de la colonne de rang r ont un poids égal à 2^r.

				$\times$	x_3	x_2	x_1	x_0
				$\times$	x_3	x_2	x_1	x_0
				$\times$	x_3	x_2	x_1	x_0
					$x_3x_0x_0$	$x_2x_0x_0$	$x_1x_0x_0$	$x_0x_0x_0$
				$x_3x_0x_1$	$x_2x_0x_1$	$x_1x_0x_1$	$x_0x_0x_1$	
				$x_3x_1x_0$	$x_2x_1x_0$	$x_1x_1x_0$	$x_0x_1x_0$	
			$x_3x_0x_2$	$x_2x_0x_2$	$x_1x_0x_2$	$x_0x_0x_2$		
			$x_3x_1x_1$	$x_2x_1x_1$	$x_1x_1x_1$	$x_0x_1x_1$		
			$x_3x_2x_0$	$x_2x_2x_0$	$x_1x_2x_0$	$x_0x_2x_0$		
		$x_3x_0x_3$	$x_2x_0x_3$	$x_1x_0x_3$	$x_0x_0x_3$			
		$x_3x_1x_2$	$x_2x_1x_2$	$x_1x_1x_2$	$x_0x_1x_2$			
		$x_3x_2x_1$	$x_2x_2x_1$	$x_1x_2x_1$	$x_0x_2x_1$			
		$x_3x_3x_0$	$x_2x_3x_0$	$x_1x_3x_0$	$x_0x_3x_0$			
	$x_3x_1x_3$	$x_2x_1x_3$	$x_1x_1x_3$	$x_0x_1x_3$				
	$x_3x_2x_2$	$x_2x_2x_2$	$x_1x_2x_2$	$x_0x_2x_2$				
	$x_3x_3x_1$	$x_2x_3x_1$	$x_1x_3x_1$	$x_0x_3x_1$				
$x_3x_2x_3$	$x_2x_2x_3$	$x_1x_2x_3$	$x_0x_2x_3$					
$x_3x_3x_2$	$x_2x_3x_2$	$x_1x_3x_2$	$x_0x_3x_2$					
$x_3x_3x_3$	$x_2x_3x_3$	$x_1x_3x_3$	$x_0x_3x_3$					

Figure 1.7 *Tableau de produits partiels pour le cube d'un entier non signé sur 4 bits.*

Afin de pouvoir distinguer les formules logiques des produits partiels implantés, on utilise pour représenter ceux-ci la notation $[\mathcal{S}]$ utilisée dans [22, p.23]. Si $\mathcal{S}$ est une proposition booléenne, alors :

$$[\mathcal{S}] = \begin{cases} 1 & \text{si } \mathcal{S} \text{ vaut vrai,} \\ 0 & \text{si } \mathcal{S} \text{ vaut faux.} \end{cases}$$

La valeur « arithmétique » de $[x_ix_j \vee x_k]$ dans la colonne de rang r est donc soit 0, soit 2^r suivant la valeur de la proposition $x_ix_j \vee x_k$. Cette notation permet l'interprétation et la manipulation d'expressions telles que $[x_ix_j \vee x_k] + [x_ix_j]$.

Une règle de réduction est l'interprétation arithmétique d'une identité logique. Les règles utilisent la notation $[\mathcal{S}]$. Par exemple, la commutativité du ET logique autorise la réduction : $[x_ix_j] + [x_jx_i] \longrightarrow 2\,[x_ix_j]$.

Une transformation est une séquence d'applications de règles de réduction. On montre plus loin que l'ordre d'application des règles lors d'une transformation est un paramètre important. On note $T_{x\rightarrow y\rightarrow z}$la transformation où la règle Rx est appliquée en priorité, suivie de la règle Ry, puis finalement Rz.

1.9.3 Travaux précédents

Pour le carré

L'optimisation d'un opérateur calculant le carré d'un nombre est un problème courant. Nous présentons ici quelques méthodes existantes. Pour mémoire, notre méthode n'apporte pas d'améliorations, dans la cas du carré, par rapport aux méthodes existantes qui utilisent les identités logiques standard.

Une méthode de calcul du carré basée sur des tables repliées est présentée dans [38]. La table des carrés est repliée sur elle-même plusieurs fois en utilisant les propriétés mathématiques de l'élévation au carré. Les résultats d'implantation en technologie CMOS montrent que cette approche apporte une amélioration importante par rapport à une implantation conventionnelle à base de ROM.

Les symétries du PPA dans l'élévation au carré peuvent aussi être utilisées dans des architectures qui calculent en série [34].

Des implantations de multiplications et d'élévation au carré sur FPGA qui utilisent les blocs multiplieurs spécialisés 18×18 bits sont proposés dans [36]. Même dans le cas d'une architecture aussi hétérogène, une élévation carré par un opérateur spécialisé demande environ moitié moins de surface qu'une multiplication.

Des unités d'élévation au carré (ou squarers) dédiées peuvent aussi être mises au point pour des entrées signées. Un squarer qui peut fonctionner au choix sur des entrées non-signées ou en complément à 2 est présenté dans [39].

Une implantation de squarers sur ASIC avec une réduction du PPA par arbres de Wallace et un additionneur final à sélection de retenue est présentée dans [33].

Une unité spécialisée d'élévation au carré demande en général moitié moins de matériel qu'une multiplication directe, et est plus rapide. Elle est aussi plus petite qu'un multiplieur avec recodage de Booth modifié, car celui-ci demande une quantité de matériel non négligeable pour le recodage du multiplicateur.

Il existe des symétries à l'intérieur du PPA lors d'une élévation à une puissance entière. Les réductions permettant de minimiser la taille et la hauteur du tableau de produits partiels pour le carré sont bien connues [11, p.221], [13, p.201]. Elles reposent sur l'application de trois règles de réduction logique qui permettent de diminuer le nombre de produits partiels ou de modifier leur répartition. L'application de ces règles transforme le PPA de telle façon qu'il donne le même résultat, mais avec un coût matériel moindre et plus rapidement.

Les deux premières règles découlent de propriétés logiques de l'opération de conjonction (ET) :

L'idempotence de la conjonction : $x_i \wedge x_i = x_i$. La réduction associée supprime une porte ET.

La commutativité de la conjonction : $x_i \wedge x_j = x_j \wedge x_i$. On peut donc remplacer certaines additions de produits partiels avec des multiplications par 2 :

$$[x_i \wedge x_j] + [x_j \wedge x_i] = 2\,[x_i \wedge x_j]$$

Le terme $2\,[x_i \wedge x_j]$ de poids r est remplacé par $[x_i \wedge x_j]$ de poids $r+1$.

Une troisième règle peut être utilisée :

$$[x_i \wedge x_j] + [x_i] = 2\,[x_i \wedge x_j] + [x_i \wedge \overline{x_j}]$$

Les deux premières règles sont appliquées autant que possible, car elles diminuent strictement le coût matériel de l'unité. La dernière règle est utilisée pour réduire la hauteur du tableau de

produits partiels [11, p.221], et peut aussi servir à diminuer la longueur de l'addition finale lors de sa somme [13, p.201].

Pour le cube

Le cube peut être facilement calculé par une multiplication $n \times n$ bits suivie d'un multiplication $2n \times n$ bits, mais il est aussi possible d'appliquer de nombreuses simplifications sur le PPA d'une unité d'élévation au cube.

On trouve dans un papier de Liddicoat et Flynn [31] l'étude d'une unité calculant le cube de l'argument. Les réductions du PPA reposent sur une extension des deux premières identités classiques utilisées pour le carré :

Idempotence de la conjonction : $x_i \wedge x_i \wedge x_i = x_i$,

Associativité et commutativité : $x_i \wedge x_j \wedge x_k = x_i \wedge x_k \wedge x_j = x_j \wedge x_i \wedge x_j = \ldots$

Après avoir utilisé les règles de réduction correspondantes, les produits partiels sont divisés en deux groupes : les PP de la forme $x_i x_i x_i = x_i$, qui apparaissent une seule fois, et ceux de la forme $x_i x_i x_j = x_i x_j$ et $x_i x_j x_k$. Ces derniers apparaissent respectivement 3 et 6 fois chacun.

Les auteurs proposent de sommer le tableau de produits partiels sous la forme $X_1 + 3 \times X_3$, où X_1 contient les produits partiels qui apparaissent en un seul endroit, et X_3 contient une fois les produits partiels qui apparaissent 3 fois (ou $6 = 3 \times 2$ fois) dans le PPA. Ce tri des produits partiels permet une forte diminution du coût matériel par rapport à une implantation directe du PPA, mais le temps de calcul dans le circuit augmente légèrement. Cette implantation reste plus coûteuse qu'un schéma à base de multiplieurs, mais est plus rapide.

Autres méthodes de calcul des puissances

Une modification d'approximation linéaire par morceaux est présentée dans [21]. Elle calcule une puissance x^p d'un opérande x. Le rationnel p est de la forme $\pm 2^k$ ou bien $\pm 2^{k_1} \pm 2^{k_2}$, où k et k_1 sont des entiers et k_2 est un entier positif. Cette méthode peut être utilisée jusqu'à 24 bits, mais aucun résultat d'implantation n'est donné.

Dans le cas d'un argument en virgule flottante, des approximations polynomiales peuvent être utilisées. Dans [32], une approximation minimax de degré 2 est utilisée. Dans ce type de solution, il existe un compromis complexe entre les performances et la précision obtenue (par exemple un arrondi fidèle).

1.9.4 Nouvelles identités et transformations

Généralisation des règles existantes

La première règle se généralise facilement, en utilisant l'idempotence de la conjonction ET :

$$[x_i \wedge x_i \cdots \wedge x_i] = [x_i]. \tag{R0}$$

La seconde réduction est une simple extension utilisant l'associativité et la commutativité de la conjonction. La sortie d'une porte ET reste la même quels que soient les permutations et regroupements appliqués à ses entrées. Par exemple, $x_1 \wedge x_2 \wedge x_3 \wedge x_4 = (x_2 \wedge x_4) \wedge (x_1 \wedge x_3)$. On en déduit la règle de réduction générale :

$$[x_i x_j \cdots x_l] + [x_{\sigma(i)} x_{\sigma(j)} \cdots x_{\sigma(l)}] = 2\,[x_i x_j \cdots x_l], \tag{R1}$$

où σ est une permutation quelconque des indices $\{i, j, \ldots, l\}$.

La troisième réduction utilisée pour les opérateurs d'élévation au carré peut s'étendre directement en :

$$[P \wedge x_i] + [P] = 2\,[P \wedge x_i] + [P \wedge \overline{x_i}],$$

où P est une proposition logique. Cette réduction permet l'économie d'un peu de matériel, mais une forme plus générale de la règle peut être dérivée.

L'idée derrière la réduction générale est d'additionner deux produits partiels P_1 et P_2, où P_1 est toujours vrai si P_2 l'est aussi (c'est à dire qu'on a $P_2 \Rightarrow P_1$). Le bit de somme vaut donc 1 lorsque P_1 est vrai et que P_2 ne l'est pas (l'inverse ne peut pas se produire), et le bit de retenue vaut 1 lorsque P_2 est vrai (car alors P_1 l'est aussi). On en déduit la règle de réduction :

$$\text{si } P_2 \Rightarrow P_1, \text{ alors } [P_1] + [P_2] = 2\,[P_2] + [P_1 \wedge \overline{P_2}] \tag{R2}$$

Exemple 1.7 $[x_1x_2x_3] + [x_1] = 2\,[x_1x_2x_3] + [x_1 \wedge (\overline{x_2} \vee \overline{x_3})]$

Cette réduction prise seule ne semble pas réduire le nombre de produits partiels. En fait, elle remplace l'addition de deux PP de la même colonne par celle de deux produits partiels dans des colonnes consécutives, ce qui revient à simplement les juxtaposer.

Exemple 1.8 *La figure 1.8 illustre une transformation qui se produit lorsqu'on réduit un PPA pour l'élévation au cube en utilisant R0, R1 et R2.*

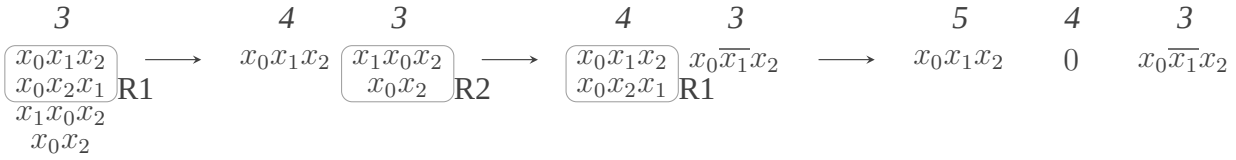

Figure 1.8 *Exemple d'une séquence de réductions dans une unité d'élévation au cube.*

Nouvelles identités et règles de réduction

La réduction R2 introduit des produits partiels où certains bits sont inversés. Dans certains cas, la somme de deux PP ne peut pas produire de retenue. Par exemple, $[x_1x_2] + [\overline{x_2}x_3] < 2$. Il est alors possible de remplacer un demi-additionneur (Half-Adder ou HA) par une simple porte OU à deux entrées. On en déduit une nouvelle règle de réduction :

$$\text{si } \overline{P_1 \wedge P_2}, \text{ alors } [P_1] + [P_2] = [P_1 \vee P_2]. \tag{R3}$$

où P_1 et P_2 sont deux propositions logiques. La condition $\overline{P_1 \wedge P_2}$ implique que P_1 et P_2 ne peuvent pas tous deux être vrais en même temps. Il n'y a donc jamais de retenue dans leur addition, et la porte XOR (ou exclusif) du HA peut être remplacée par une simple porte OR.

Une autre règle de réduction plus spécifique peut être utilisée, qui est basée sur la loi du tiers exclu :

$$[P \wedge x_i] + [P \wedge \overline{x_i}] = [P], \tag{R4}$$

où P est une proposition logique. L'application de cette règle en priorité sur R3 permet à d'autres réductions d'être appliquées plus tard.

1.9.5 Optimisation des transformations

La figure 1.8 montre que les produits partiels générés par une réduction peuvent être réutilisés au cours des réductions suivantes. En conséquence, l'ordre d'application des règles de réduction au cours de la transformation est un problème majeur si on veut minimiser la surface du circuit. Ce problème n'est pas couvert par les travaux existants. Il n'est pas possible en général de calculer la séquence optimale de réductions dans le PPA. On va plutôt fonctionner en donnant différentes priorités aux règles de réduction. Une transformation sera caractérisée par cet ordre de priorités.

rang : *8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
		x_2	x_1x_2	x_0x_2	$x_0x_1x_2$	x_0x_2	x_0x_1	x_0
			x_1x_2	x_1x_2	$x_0x_2x_1$	x_0x_1	x_1x_0	
			x_2x_1	x_2x_1	$x_1x_0x_2$	x_2x_0	x_1x_0	
				x_0x_2	$x_2x_1x_0$	x_0x_1		
				x_2x_1	$x_1x_2x_0$	x_1x_0		
				x_2x_0	$x_2x_0x_1$	x_2x_0		
					x_1			
$x_0x_1x_2$	$x_1x_2\overline{x_0}$	$x_2(\overline{x_1} \vee \overline{x_0})$	x_0x_2	$x_2(x_1 \vee x_0)$	x_0x_2	x_2x_0	x_1x_0	x_0
				x_0x_1	x_1			
$x_1x_0x_2$	$x_2x_1\overline{x_0}$	x_2	$x_2\overline{x_1}x_0$	$x_2x_0 \vee x_1x_0$ $\vee x_1x_2$	$\overline{x_2}x_1 \vee \overline{x_0}x_1$ $\vee x_0x_2\overline{x_1}$	x_0x_2	x_0x_1	x_0

Figure 1.9 *Tableaux de produits partiels pour un cube sur $w = 3$ bits avant réduction, après réduction par $T_{0\to1\to2\to4\to3}$ et après réduction par $T_{0\to2\to4\to3\to1}$.*

La figure 1.9 illustre la différence entre deux types de transformations pour le PPA d'un cube calculé sur 3 bits. En haut de la figure, on voit le tableau de produits partiels avant réductions. Le bas de la figure montre d'abord le PPA transformé par $T_{0\to1\to2\to4\to3}$. Dans cette transformation, on applique d'abord R0 autant que possible, puis R1, puis R2, etc. Si en appliquant R2 on fait apparaître de nouveaux PP pour lesquels on peut utiliser R1, alors on applique R1 en priorité. On continue à appliquer les règles par priorité décroissante jusqu'à ce qu'aucune réduction ne

soit possible. Le second PPA transformé est obtenu par $T_{0\to2\to4\to3\to1}$. On constate que le PPA obtenu par application de $T_{0\to1\to2\to4\to3}$ a besoin de 5 cellules d'additions (Full adder ou FA) et une cellule HA pour être sommé, alors que $T_{0\to2\to4\to3\to1}$ donne une solution sans additions, bien que les produits partiels soient plus compliqués.

Les figures 1.10 et 1.11 détaillent la réduction de la colonne de rang 3 de la figure 1.9, en utilisant les transformations $T_{0\to1\to2\to4\to3}$ et $T_{0\to2\to4\to3\to1}$ respectivement. La colonne de rang 3 est celle obtenue après réductions des colonnes de rang < 3.

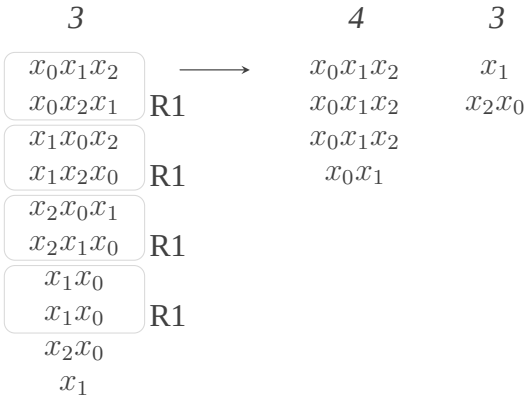

Figure 1.10 *Réduction de la colonne de rang 3 à l'aide de la transformation $T_{0\to1\to2\to4\to3}$.*

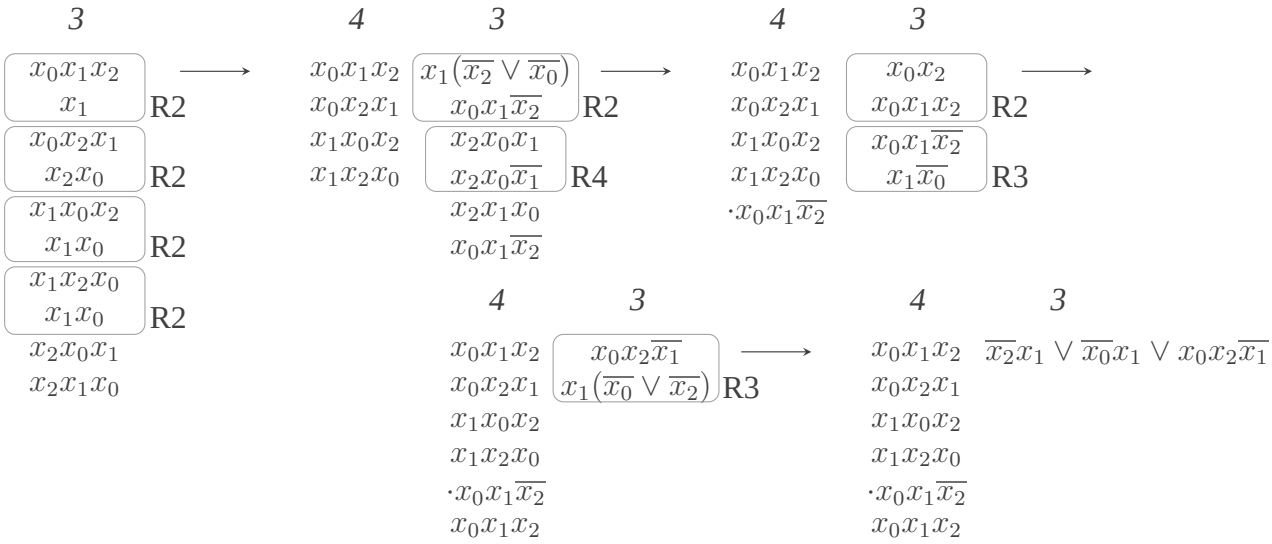

Figure 1.11 *Réduction de la colonne de rang 3 à l'aide de la transformation $T_{0\to2\to4\to3\to1}$.*

On constate que chaque règle prend deux produits partiels dans la colonne de rang r et introduit au plus un produit aux rangs r et $r+1$. Une bonne façon d'appliquer les règles est donc de transformer le PPA depuis les colonnes de poids faibles vers les poids forts. Ceci permet d'avoir un maximum de produits partiels disponibles pour la réduction de chaque colonne. On va « propager » les produits partiels réduits vers la gauche.

Chaque fois qu'un couple de produits partiels est réduit, un nouveau produit partiel peut être réintroduit dans la colonne courante. Il faut donc réessayer les réductions. On peut réduire de

transformation	$w = 16$		$w = 24$	
	#PP	ratio	#PP	ratio
$T_{0\to2\to4\to3\to1}$	997	100%	3723	100%
$T_{0\to4\to3\to1\to2}$	1090	109%	3916	105%
$T_{0\to2\to1\to4\to3}$	1127	113%	4071	109%
$T_{0\to1\to2\to4\to3}$	1134	114%	4062	109%
$T_{0\to4\to3\to2\to1}$	1465	147%	4892	131%

Table 1.3 *Nombre de produits partiels dans le tableau réduit pour différents ordres d'application des règles.*

plusieurs ordres le temps pris par la transformation en gardant une trace des produits partiels déjà testés pour les réductions.

Différent ordres d'applications ont été testés. Une transformation évidente est celle obtenue en étendant les méthodes classiques, et où les nouvelles règles sont appliquées après R1 et R2. C'est un choix raisonnable, dans la mesure où R1 est la règle la plus efficace en termes de réductions de matériel, et où les nouvelles règles R3 et R4 reposent sur les bits inversés introduits par l'application de R2. La transformation obtenue est $T_{0\to1\to2\to4\to3}$.

Pourtant, l'ordre le plus efficace que nous avons trouvé lors des tests est $T_{0\to2\to4\to3\to1}$. En appliquant d'abord la réduction R2, on introduit plus de bits inversés qui sont ensuite utilisés par les nouvelles règles. Les produits partiels identiques qui restent sont ensuite simplifiés par R1. Cet ordre permet d'avoir moins de produits partiels dans le PPA transformé. Ils sont par contre plus compliqués (les fonctions logiques dépendent de plus de bits de x), mais ceci n'a que peu d'influence dans une implantation sur un FPGA à base de tables look-up.

La table 1.3 donne le nombre de produits partiels restant dans un PPA calculant le cube d'un opérande de 16 ou 24 bits pour différents ordres d'application des règles. On constate que l'ordre d'application des règles est important si on veut minimiser le nombre de produits partiels duquel dépend la taille de l'opérateur. La transformation $T_{0\to2\to4\to3\to1}$ semble bien être la plus efficace.

1.9.6 Résultats pratiques

Des opérateurs ont été synthétisés en utilisant la méthode proposée pour le calcul du cube avec des largeurs d'entrée $w \in \{4, 8, 12, 16, 20, 24\}$ bits. L'implantation a été faite sur un FPGA Spartan3 (S1500) de Xilinx à l'aide des outils ISE8.1. La synthèse est effectuée avec un effort orienté en surface, et le placement-routage est fait avec un effort d'optimisation standard.

La table 1.4 donne les résultats d'implantation obtenus en se basant sur les méthodes de réduction existantes. La table de gauche utilise seulement les règles R0 et R1 ($T_{0\to1}$). Ceci correspond aux résultats des méthodes classiques [11, p.221] et [13, p.201]. La partie droite est une simple extension des règles de réductions connues.

La table 1.5 présente les résultats d'implantation obtenus par notre méthode pour deux ordres différents. La partie gauche contient les résultats pour la transformation $T_{0\to1\to2\to4\to3}$ et la partie droite donne ceux obtenus en appliquant $T_{0\to2\to4\to3\to1}$. On constate que $T_{0\to2\to4\to3\to1}$ donne en général les meilleurs résultats, même après implantation.

La figure 1.12 résume ces résultats pour une unité élevant au cube un argument d'une largeur de $w \in \{4, 8, 12, 16, 20, 24\}$ bits. La réduction est moins efficace pour les grandes tailles d'entrée. Nous pensons que la structure du FPGA utilisé en est responsable. Une amélioration plus constante

$T_{0\to1}$				
w	#PP	Taille (tranches)	Délai (ns)	Cellules d'addition
4	30	22	9.4	22
8	218	153	13.5	202
12	694	527	16.8	670
16	1586	1223	19.7	1554
20	3022	2292	21.3	2982
24	5130	3822	24.5	5082

$T_{0\to1\to2}$				
w	#PP	Taille (tranches)	Délai (ns)	Cellules d'addition
4	26	21	8.5	17
8	182	139	15.3	166
12	594	519	16.4	577
16	1390	1209	18.2	1362
20	2698	2145	21.1	2666
24	4646	3799	24.3	4613

Table 1.4 *Résultats d'implantation des méthodes classiques ([11, p. 221] et [13, p. 201]).*

$T_{0\to1\to2\to4\to3}$				
w	#PP	Taille (tranches)	Délai (ns)	Cellules d'addition
4	20	16	6.7	10
8	126	120	11.8	110
12	454	445	14.9	436
16	1134	1128	18.4	1106
20	2294	2060	21.7	2265
24	4062	3750	23.2	4030

$T_{0\to2\to4\to3\to1}$				
w	#PP	Taille (tranches)	Délai (ns)	Cellules d'addition
4	17	15	7.1	9
8	111	119	14.9	95
12	393	463	15.3	377
16	997	1068	18.6	971
20	2058	1991	21.5	2030
24	3723	3540	23.3	3693

Table 1.5 *Résultats d'implantation pour notre méthode*

est probable sur des circuits de type ASIC.

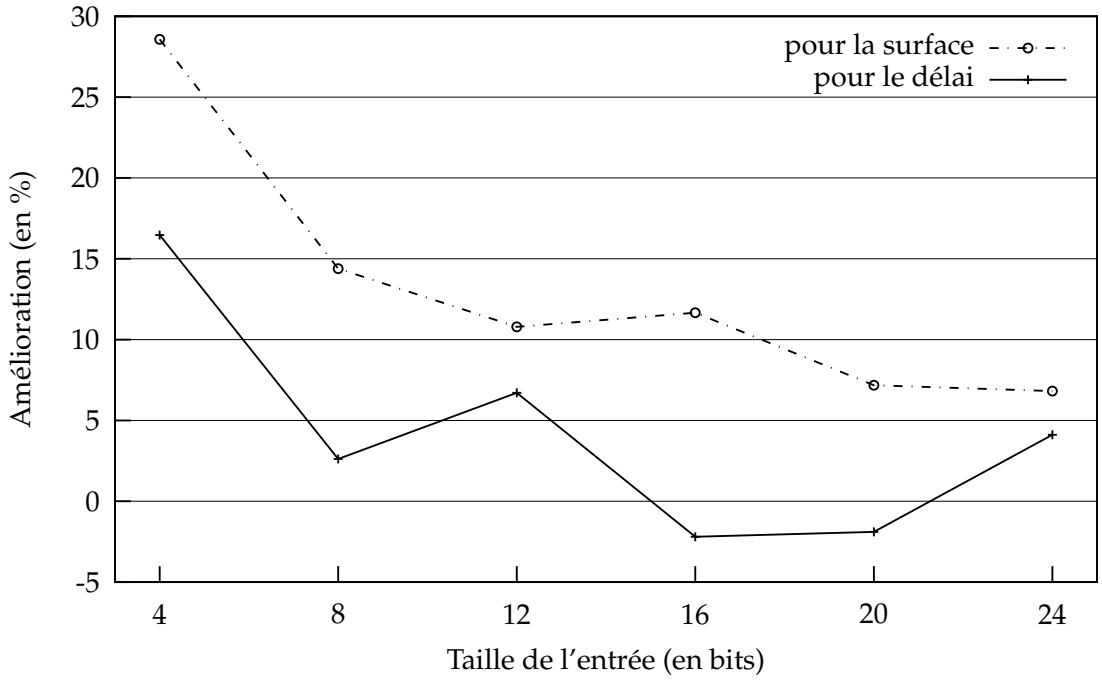

Figure 1.12 *Améliorations en surface et vitesse obtenus par notre méthode par rapport aux méthodes de réduction classiques, en fonction de la largeur de l'entrée, pour un opérateur calculant le cube.*

1.9.7 Améliorations ultérieures

Les opérateurs pour les puissances ont bénéficié de plusieurs améliorations depuis ceux présentés lors de la conférence SPIE [4].

Tout d'abord, les arbres d'additions qui servent à sommer le tableau de produits partiels étaient jusqu'ici générés par le synthétiseur. Un algorithme a été ajouté, il décrit en VHDL des arbres d'additions optimisés en taille.

D'autre part, les opérateurs pour les puissances sont désormais obtenus par une méthode qui se base à la fois sur les transformations présentées plus haut et sur une idée de Liddicoat et Flynn [31].

Addition du tableau de produits partiels

Suite à des expérimentations, il apparaît que les outils de synthèse de la suite ISE ne sont pas toujours capables d'organiser efficacement sous forme d'arbre des séries d'additions multiples. C'est le cas en particulier pour celles qui correspondent à la sommation d'un PPA. Ces additions contiennent de nombreux termes creux qui sont dus à l'irrégularité du PPA. On retrouve le même problème dans les tableaux de produits partiels des unités square-and-multiply, et dans l'addition des estimations $\Delta^i(x, c_i)$ décalées pour les évaluations polynomiales.

Une fonction a donc été ajoutée à la partie du programme qui génère la description VHDL du circuit. Lorsque plusieurs termes doivent être additionnés, un exploration heuristique est faite sur les différentes façons d'agencer l'arbre d'addition, dans le but de minimiser à la fois le nombre de cellules d'additions utilisées et la profondeur de l'arbre. Dans le cas d'implantations sur FPGA, on essaie d'utiliser au mieux les additions à propagation de retenue, qui bénéficient de la ligne de *fast-carry* disponible sur les circuits étudiés. Une exploration complète des agencements possibles se révèle en pratique trop longue, même pour un petit nombre d'additions.

Le principe de l'algorithme utilisé est de réduire autant que possible le nombre de bits qui restent à additionner en « payant » un minimum de cellules d'additions. À chaque itération, une fonction de coût est donc assignée aux additions possibles, qui dépend du nombre de bits réduits (additionnés), du nombre de cellules nécessaires et de « l'âge » des termes additionnés (on essaie d'équilibrer l'arbre en additionnant en priorité les termes générés le plus tôt).

Les arbres d'additions obtenus à l'aide de cet algorithme sont réduits en moyenne de 20 à 25%, par rapport aux mêmes sommations où l'on laisse le synthétiseur faire seul.

Tri des produits partiels

La méthode présentée utilise des réductions logiques pour minimiser le nombre de produits partiels dans le tableau de produits partiels. Ceci a pour conséquence une réduction du coût matériel des opérateurs.

Une autre approche évoquée plus haut est celle de Liddicoat et Flynn [31]. Leur étude porte sur une unité calculant le cube de l'argument. Les produits partiels du PPA y sont divisés en deux groupes : ceux de la forme $x_i x_i x_i = x_i$, qui apparaissent une seule fois, et ceux de la forme $x_i x_i x_j = x_i x_j$ et $x_i x_j x_k$. Ces derniers apparaissent 3 et 6 fois chacun. Les auteurs somment le tableau de produits partiels sous la forme $X_1 + 3 \times X_3$, où X_1 contient les produits partiels qui apparaissent en un seul endroit, et X_3 contient une fois les produits partiels qui apparaissent 3 fois (ou $6 = 3 \times 2$ fois) dans le PPA.

Ce tri des produits partiels permet de diminuer de manière importante le coût matériel des opérateurs. En moyenne, les opérateurs utilisant l'idée de Liddicoat et Flynn sont 15% plus petits

taille de l'entrée	Tri des produits partiels taille (tranches)	délai (ns)	Méthode hybride taille (tranches)	délai (ns)
8	129	24.4	119	20.5
12	373	47.1	359	39.3
16	806	75.7	782	64.6
20	1441	109.9	1422	96.4
24	2345	149.9	2335	133.9

Table 1.6 *Résultats d'implantation pour une élévation exacte au cube.*

que ceux obtenus par notre méthode de réduction présentée à SPIE'05. Par contre ils sont plus lents.

Méthode hybride pour le calcul des puissances

Il est possible d'obtenir des opérateurs à la fois plus petits et plus rapides en combinant les transformations logiques et le tri des produits partiels. Le procédé qui permet d'obtenir les opérateurs les plus petits applique les étapes suivantes :

1. Appliquer la transformation $T_{0\rightarrow 1}$ à travers le tableau de produits partiels.

2. Trier les produits partiels dans quatre sous-PPA X_1, X_3, X_9 et X_{15} de telle façon que la somme $\sum iX_i$ donne le même résultat que le PPA de départ. Par exemple, deux produits partiels avec un poids total de 9×2^r (ils apparaissent aux rangs r et $r+3$) du PPA vont être remplacés par un seul produit partiel identique de rang r dans X_9.

3. Appliquer la transformation $T_{0\rightarrow 2\rightarrow 4\rightarrow 3\rightarrow 1}$ aux sous-PPA triés X_1, X_3, X_9 et X_{15}.

4. Si des réductions ont été appliquées, on fusionne les sous-PPA et on recommence à partir de l'étape **1**. Sinon on retourne les sous-PPA triés.

L'opérateur obtenu par cette méthode calcule x^n sous la forme :

$$X_1 + (2^1 + 2^0) \times X_3 + (2^3 + 2^0) \times X_9 + (2^4 - 2^0) \times X_{15}$$

La figure 1.13 illustre les étapes de la réduction d'une unité qui calcule un cube sur 8 bits. Le nombre de produits partiels dans les tableaux X_1, X_3, X_9 et X_{15} est clairement inférieur à celui du PPA du dessus, réduit sans tri des produits partiels. L'opérateur obtenu sera donc plus petit.

Les résultats de synthèse, rapportés dans la table 1.6 montrent que la méthode hybride obtient des opérateurs de 1 à 3% plus petits, et 11 à 17% plus rapides que le tri des PP seul. L'écart en taille devient plus grand pour une élévation à des puissances supérieures, mais un calcul exact n'a alors plus beaucoup de sens.

Opérateurs tronqués

Lorsque l'on tronque les tableaux de produits partiels comme c'est le cas pour les approximations polynomiales de ce chapitre, on constate que la méthode hybride présente un avantage par rapport au tri des produits partiels seul. En effet, elle demande une estimation sur un plus petit nombre de colonnes pour atteindre la même précision. Pour le même nombre de colonnes implantées, elle permet aussi d'atteindre une précision supérieure.

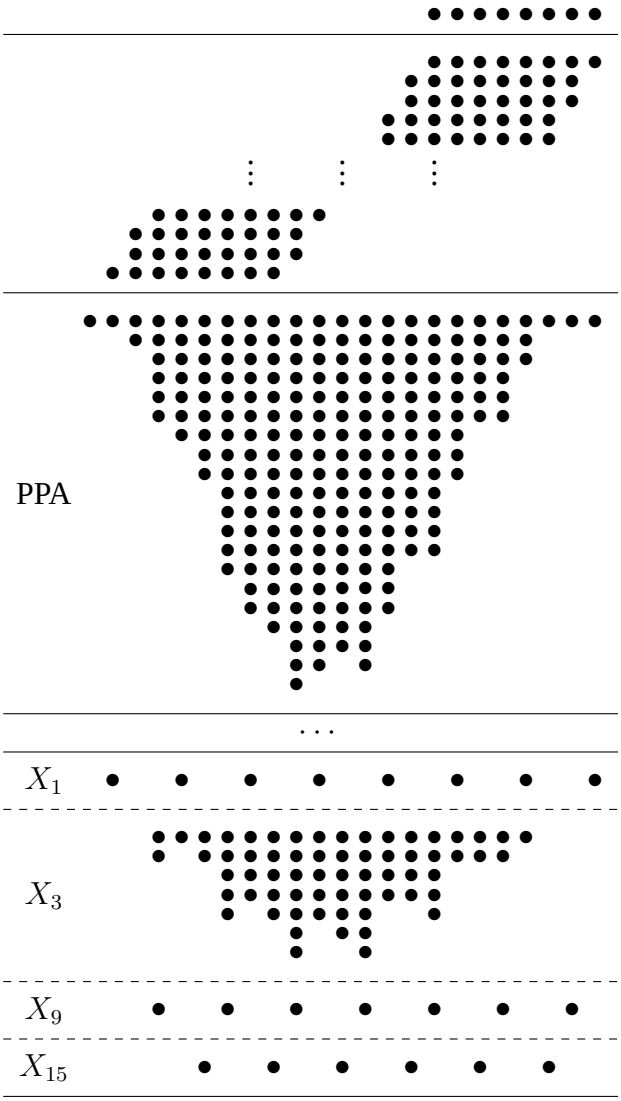

Figure 1.13 *Réduction du tableau de produits partiels calculant le cube d'un argument de 8 bits.*

La raison est que la transformation $T_{0\to2\to4\to3\to1}$, en plus de diminuer le nombre de produits partiels dans un tableau, fait « migrer » des produits partiels vers la partie du PPA correspondant aux poids forts. En conséquence, pour un même nombre de colonnes implantées, la méthode hybride prend en compte plus de termes, donc atteint une meilleure précision. Cette propagation logique explique aussi les délais moindres dans les unités hybrides.

Exemple 1.9 *Pour l'exemple présenté d'un opérateur évaluant la fonction sinus sur 12 bits à l'aide d'une approximation polynomiale de degré 3, on estime x^3 sur 9 colonnes.*
Avec la méthode hybride, on a obtenu deux opérateurs qui offrent pour $(n_1, n_2, n_3) = (1, 1, 2)$ une précision moyenne supérieure à 12.1 bits et pour $(n_1, n_2, n_3) = (1, 1, 3)$ une précision moyenne supérieure à 12.6 bits.
En triant simplement les produits partiels (sans appliquer de réductions) et dans les mêmes conditions, on aboutit à deux autres solutions : pour $(n_1, n_2, n_3) = (1, 1, 2)$ une précision moyenne supérieure à 11.8 bits et pour $(n_1, n_2, n_3) = (1, 1, 3)$ une précision moyenne supérieure à 12 bits.

> *On arrive à gagner plus d'un demi-bit de précision moyenne pour le même nombre de colonnes implantées.*

Les figures 1.14, 1.15 et 1.16 montrent l'amélioration en terme d'erreur d'estimation qu'apporte la méthode hybride par rapport à un tri des produits partiels seul. Pour un argument sur 16 bits, ils montrent en % la diminution d'erreur obtenue, lorsque l'estimation du résultat est faite à l'aide du même nombre de colonnes implantées. La figure 1.14 montre qu'on obtient des améliorations ponctuelles pour un opérateur calculant un cube. Pour une élévation à la puissance 4 (figure 1.15), la diminution de l'erreur est en moyenne de 15%. Enfin, pour une puissance de 5 (figure 1.16) l'erreur faite lors de l'estimation est réduite de moitié.

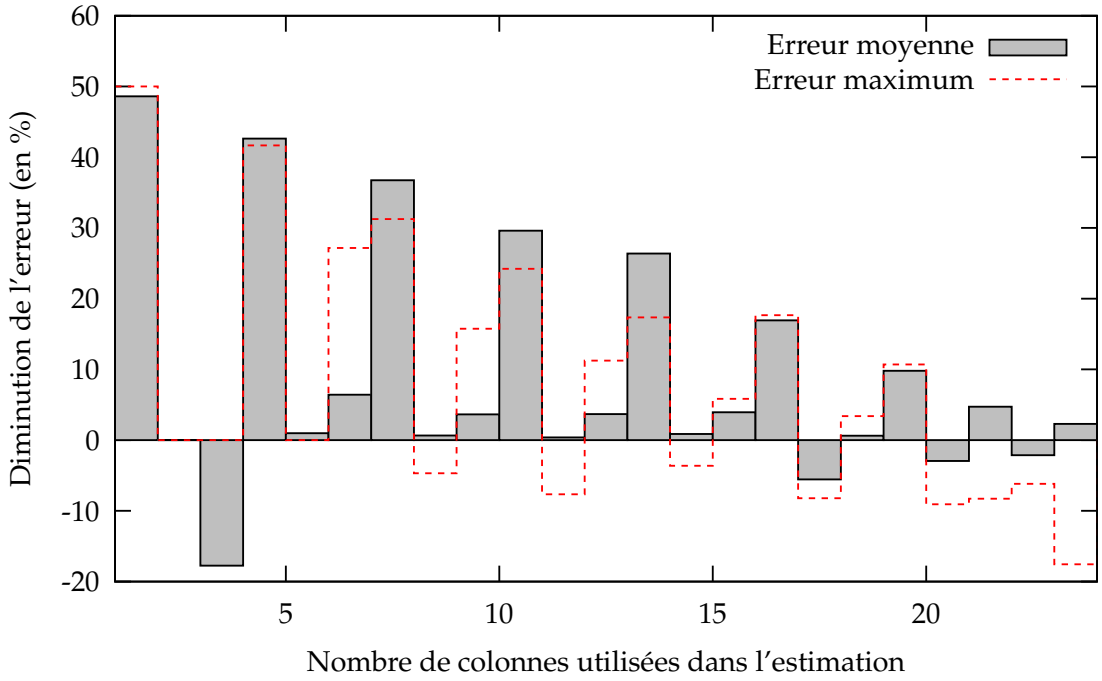

Figure 1.14 *Diminution de l'erreur en % en fonction du nombre de colonnes implantées dans l'estimation, en utilisant un opérateur hybride par rapport à un tri des produits partiels seul, pour un argument de 16 bits élevé à la puissance 3*

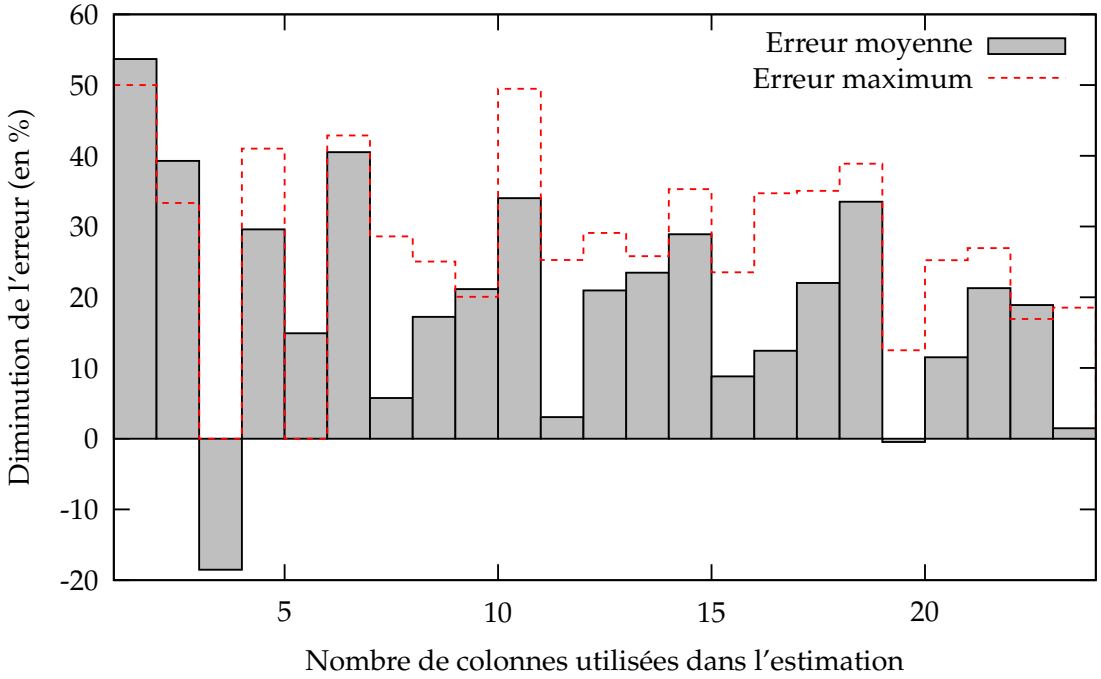

Figure 1.15 *Diminution de l'erreur en % en fonction du nombre de colonnes implantées dans l'estimation, en utilisant un opérateur hybride par rapport à un tri des produits partiels seul, pour un argument de 16 bits élevé à la puissance 4*

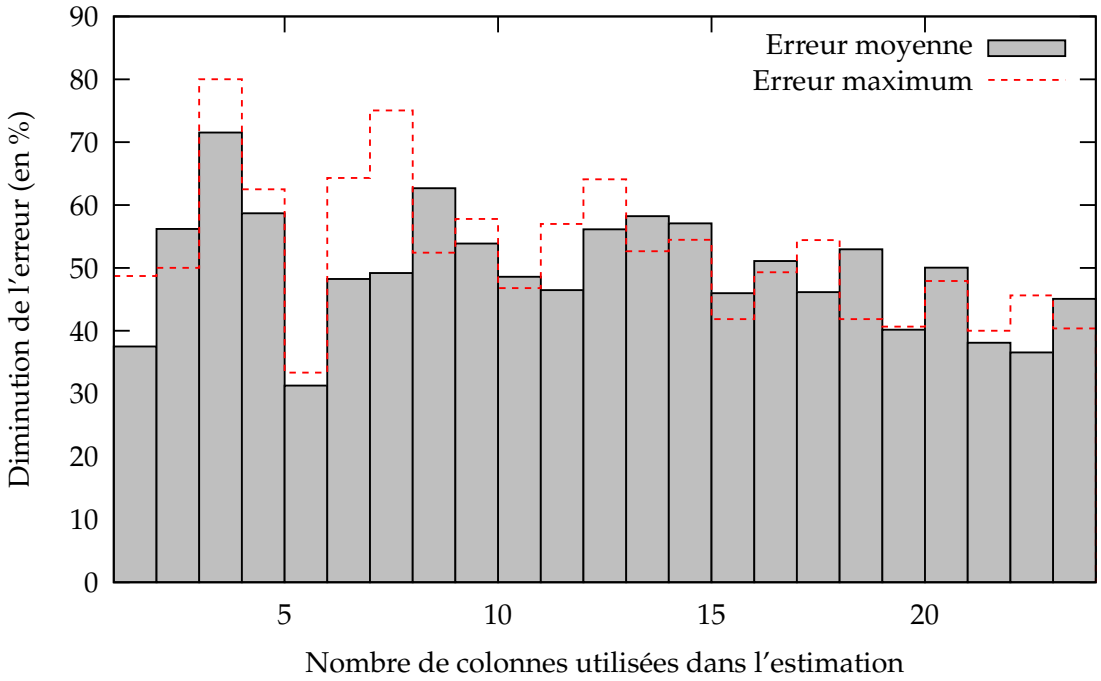

Figure 1.16 *Diminution de l'erreur en % en fonction du nombre de colonnes implantées dans l'estimation, en utilisant un opérateur hybride par rapport à un tri des produits partiels seul, pour un argument de 16 bits élevé à la puissance 5*

CHAPITRE 2

Génération automatique d'opérateurs à récurrence de chiffres

2.1 Introduction

Certaines fonctions algébriques sont utiles dans les circuits intégrés modernes tels que les systèmes sur puce pour les télécommunications et les applications du multimédia. Même lorsqu'elles ne sont pas utilisées fréquemment, leur temps de calcul peut influer de manière significative sur les performances globales. Par exemple, la division et la racine carrée sont les opérations arithmétiques de base les moins fréquemment utilisées, mais aussi les plus compliquées. Des mesures réalisées dans [45] sur les benchmarks SPECfp montrent que 40% de la latence y est due aux divisions et racines carrées en virgule flottante, alors qu'elles ne représentent que 3% des opérations flottantes réalisées. Ceci montre qu'une implantation matérielle pour l'évaluation à haute performance des fonctions algébriques peut être d'une importance critique.

La division et la racine carrée sont les fonctions algébriques que l'on rencontre principalement. Cependant, d'autres fonctions peuvent être implantées en matériel pour accélérer les applications courantes. Par exemple, l'inversion est récurrente dans la démodulation de fréquence. La norme euclidienne 3D $\sqrt{x^2+y^2+z^2}$ est primordiale dans les applications graphiques sur ordinateur. Son calcul à l'aide d'élévations au carré, puis d'additions suivies d'une racine carrée introduit un délai très long et demande la gestion des erreurs d'arrondi intermédiaires, ce qui complique encore les choses.

Les algorithmes dits *à récurrence de chiffres*, ou algorithmes *à additions et décalages*, évaluent une fonction en produisant à chaque itération un nombre fixe de bits du résultat [11]. À chaque itération, on obtient un chiffre du résultat (qui correspond à un ou plusieurs bits), en partant des chiffres ayant le poids le plus fort. Ces algorithmes sont similaires aux méthodes « à la main » pour la division et l'extraction de la racine carrée. Leur implantation ne demande que des décalages, additions et soustractions, et quelques petites tables pour le calcul de chaque itération.

Une classe de ces algorithmes, celle des algorithmes SRT, a été découverte indépendamment par D.W. Sweeney, J.E. Robertson [48] et K.B. Tocher [50] vers la fin des années 50. En fournissant un résultat écrit dans une représentation redondante, ils peuvent corriger une petite « erreur » faite dans le choix d'un nouveau chiffre du résultat au cours des itérations suivantes. Ceci permet un choix du chiffre plus simple, qui leur donne un avantage en termes de coût matériel et de vitesse. Les opérateurs SRT peuvent évaluer les fonctions algébriques usuelles, et une extension simple de la classe SRT autorise le calcul de certaines fonctions élémentaires comme l'exponentielle et le

logarithme [12]. Un livre entier [51] est dédié au calcul de la division et à l'extraction de la racine carrée à l'aide d'algorithmes SRT.

Le nombre de paramètres est trop important pour qu'il soit possible de donner une description générique d'un algorithme à récurrence de chiffres optimisé pour n'importe quelle fonction. Même dans le cas simple de la division et de la racine carrée et pour un jeu de paramètres donné, les formules analytiques décrivant un opérateur sont compliquées [43]. On ne sait pas si ces formules existent dans tous les cas. Ces algorithmes sont complexes et leur mise au point résulte d'un processus long et sujet à erreur, à la fois au niveau mathématique et lors de la traduction en circuit. Par exemple, le fameux bug dans l'unité de division flottante du Pentium était dû à quelques entrées incorrectes dans la table de sélection des chiffres du quotient [40].

De nombreux opérateurs dédiés à une fonction, ou à un jeu de fonctions spécifiques ont été développés par le passé. Par exemple, un opérateur calculant l'inverse de la racine carrée est présenté dans [53]. Un algorithme pour l'évaluation de la norme euclidienne en trois dimensions est étudié dans [49]. Des algorithmes permettant de calculer plusieurs fonctions dans un même opérateur ont aussi fait l'objet de publications, par exemple pour la division et la racine carrée [52], ou plus récemment pour la division, la racine carrée et la racine carrée inverse [44]. Les travaux qui traitent de ces algorithmes laissent souvent de côté les aspects pratiques, en présentant l'algorithme pour un jeu fixé de paramètres sans donner de détails ou de résultats pour l'implantation. La complexité du processus et l'absence de descriptions matérielles font de ces opérateurs une solution peu intéressante pour les non spécialistes.

Le but de nos travaux est la conception et l'implantation d'algorithmes pour la génération automatique d'opérateurs matériels évaluant les fonctions algébriques couramment utilisées : division, racine carré, norme euclidienne, etc. Bien que difficile à écrire et à vérifier, un générateur automatique est capable de manipuler finement des bornes et des erreurs qui sont habituellement négligées dans une étude faite manuellement, car très dures à gérer. Ceci permet en pratique d'obtenir facilement des opérateurs petits et rapides. Un premier générateur codé en C++ et nommé `Divgen` permet d'obtenir des descriptions en VHDL de diviseurs pour plusieurs algorithmes existants, avec des « options » qui consistent en diverses méthodes d'optimisation. Nous étendons actuellement ce générateur pour générer des opérateurs calculant d'autres fonctions algébriques, avec la possibilité d'effectuer de nouvelles optimisations. Nous souhaitons aussi fournir en parallèle de la description VHDL des outils pour la validation des opérateurs générés (composant logiciel équivalent, banc de test pour les simulations VHDL, démonstration en LaTeX).

Ce chapitre est organisé de la façon suivante. Les notations utiles et quelques jalons sont présentés dans la section 2.2. La section 2.3 décrit quelques algorithmes à récurrence de chiffres existants pour la division. Nous présentons ensuite en section 2.4 le programme `Divgen` qui permet d'obtenir facilement des descriptions VHDL optimisées de diviseurs. Cette description est suivie d'une étude fine du cas de la racine carrée en section 2.5. Ceci nous permet d'introduire la généralisation qui permet d'évaluer une classe de fonctions algébriques. Les premiers algorithmes pour une génération optimisée d'opérateurs SRT sont présentés en section 2.6. Il s'agit d'un travail en cours.

2.2 Notations et notions utiles

Au cours de ce chapitre, les notations utilisées sont celles couramment utilisées dans la communauté arithmétique, en particulier dans les travaux sur la division et la racine carrée de Ercegovac et Lang [51, 11].

Les algorithmes étudiés dans ce chapitre sont dits à récurrence de chiffres. Le résultat est représenté en base r, et un chiffre de celui-ci est obtenu à chaque itération. De nombreux paramètres sont impliqués, et la qualité d'une solution dépend des contraintes particulières à chaque implantation (par exemple le type de circuit ASIC ou FPGA), ce qui rend l'espace de recherche immense.

Dans le cas de la division x est le *dividende*, d le *diviseur*, q le *quotient* et w le *reste*. L'opération de division est définie sur les entiers par :

$$x = d \times q + w,$$

avec la contrainte $0 \leq |w| < |d|$ et signe$(d) =$ signe(w). L'extension au format virgule fixe est immédiate.

Lorsque l'on travaille sur un nombre v en virgule fixe, LSB(v) retourne le poids du LSB de v.

La base r utilisée est en général 10 ou une puissance de 2. Chaque itération calcule un nouveau chiffre du quotient noté q_j en base r, à l'itération j. On peut essayer d'accélérer le calcul en augmentant r : on obtient alors $log_2 r$ bits du quotient par itération. Par contre, le calcul fait à chaque itération devient plus compliqué, et demande plus de temps. Il est donc possible que le temps de calcul global augmente.

La notation $v[j]$ représente la valeur de v à l'itération j. Lorsque l'on utilise une estimation de la valeur v, on note celle-ci $\hat{v}$.

Pour déterminer le chiffre q_j, on étudie les valeurs du diviseur d et du reste partiel $w[j-1]$ de l'itération $j-1$. Le choix est fait par la *fonction de sélection* : $q_j =$ Sel$(d, w[j-1])$. Cette fonction calcule chaque chiffre de façon à ce que le reste partiel reste *borné*. Le reste partiel est défini par récurrence par $w[j] = rw[j-1] - dq_j$. La valeur initiale $w[0]$ est égale à x divisé par une puissance de 2 telle que les bornes soient respectées à l'itération $j = 0$. Les bornes sont calculées de façon à assurer la convergence du *résultat partiel* $q[j]$ vers le quotient. Le résultat partiel à l'itération j est défini par :

$$q[j] = \sum_{i=1}^{j} q_i r^{j-i},$$

et le quotient final est $q[n]$, où n est le nombre d'itérations nécessaires pour atteindre la précision de la sortie sur laquelle est lue le résultat. On introduit, pour noter les tailles en bits du dividende, du diviseur et du quotient respectivement, les paramètres `x_size`, `d_size` et `q_size`. On a alors :

$$n = \left\lceil \frac{\texttt{q\_size}}{\log_2 r} \right\rceil.$$

Afin d'accélérer les itérations du calcul, on utilise souvent une représentation redondante des nombres. On travaille alors en base r sur un ensemble symétrique de chiffres consécutifs $\{-\alpha, -\alpha+1, \ldots, \alpha\}$. Pour être redondante, cette représentation doit satisfaire $\alpha \geq \lceil \frac{r}{2} \rceil$. En pratique, on impose aussi la limite $\alpha \leq r-1$. On définit ρ le facteur de redondance de la base, qui vaut :

$$\frac{1}{2} < \rho = \frac{\alpha}{r-1} \leq 1.$$

Pour éviter les confusions entre les chiffres négatifs et le signe de la soustraction, on note les chiffres négatifs en les surmontant d'une barre ($-5 = \overline{5}$). Un système de représentation redondant est caractérisé par la notation r–α.

Exemple 2.1 *4–2 représente le système de numération redondante de base 4 qui utilise les chiffres de l'ensemble* $\{\overline{2}, \overline{1}, 0, 1, 2\}$*, avec un facteur de redondance* $\rho = \frac{2}{3}$*. 4–3 représente celui utilisant l'ensemble de chiffres* $\{\overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3\}$*, avec* $\rho = 1$*.*

2.3 Algorithmes de division

2.3.1 Division « à la main »

Dans la division apprise à l'école on travaille en représentation non redondante, avec $r = 10$ et des chiffres pris dans l'ensemble $\{0, 1, \ldots, 9\}$.

Exemple 2.2 *On effectue la division de* $x = 137$ *par* $d = 1093$ *:*

$$
\begin{array}{lr|l}
w[0] = & 137\mathbf{0} & 1093 \\
w[1] = & 277\mathbf{0} & \hline \\
w[2] = & 584\mathbf{0} & \mathbf{0}.1\,2\,5\,3\,4\,3\ \ldots \\
\vdots & 375\mathbf{0} & \downarrow\ \downarrow\ \downarrow \\
 & 471\mathbf{0} & q_1\ q_2\ q_3 \\
 & \ldots &
\end{array}
$$

Les **0** *en gras résultent d'un décalage dans la base (une multiplication par 10).*

On pose le reste partiel initial $w[0] = x$. On trouve ensuite un entier $q_1 = 1$ tel que $w[1] = 10w[0] - q_1 d$ reste positif et soit strictement inférieur à d. Pour toutes les étapes $0 < j \leq n$, on doit choisir un chiffre q_j aussi grand que possible mais qui respecte la condition $10w[j-1] \geq dq_j$. C'est à dire que l'on veut que $w[j] = 10w[j-1] - dq_j$ soit toujours compris dans l'intervalle $[0, d[$.

La sélection du chiffre q_j demande donc de comparer le reste partiel décalé $10w[j-1]$ aux valeurs d, $2d$, …, jusqu'à $9d$. Ces comparaisons sont coûteuses en matériel, puisque chacune demande une soustraction à propagation de retenue entre le reste partiel et un multiple du diviseur. C'est d'ailleurs d'où vient la difficulté de la division à la main : « deviner » le bon chiffre du quotient à chaque itération.

2.3.2 Division restaurante

Ce choix devient plus facile dans le cas de la même division en base $r = 2$, avec les chiffres $\{0, 1\}$. L'algorithme correspondant est appelé un algorithme de *division restaurante*. À chaque itération, on soustrait le diviseur au reste partiel multiplié par la base (cette multiplication par 2 consiste en un simple décalage, on «abaisse» un zéro). Le résultat devient le nouveau résultat partiel. S'il est positif ou nul, le nouveau chiffre du quotient est 1, sinon le nouveau chiffre devient 0 et on restaure la valeur du reste partiel (en lui ajoutant le diviseur d). Habituellement, cette restauration n'est pas effectuée par une véritable addition mais en stockant la valeur $2w[j-1]$ au lieu de $w[j]$. Ceci permet de remplacer une addition dans le chemin critique par un simple multiplexeur.

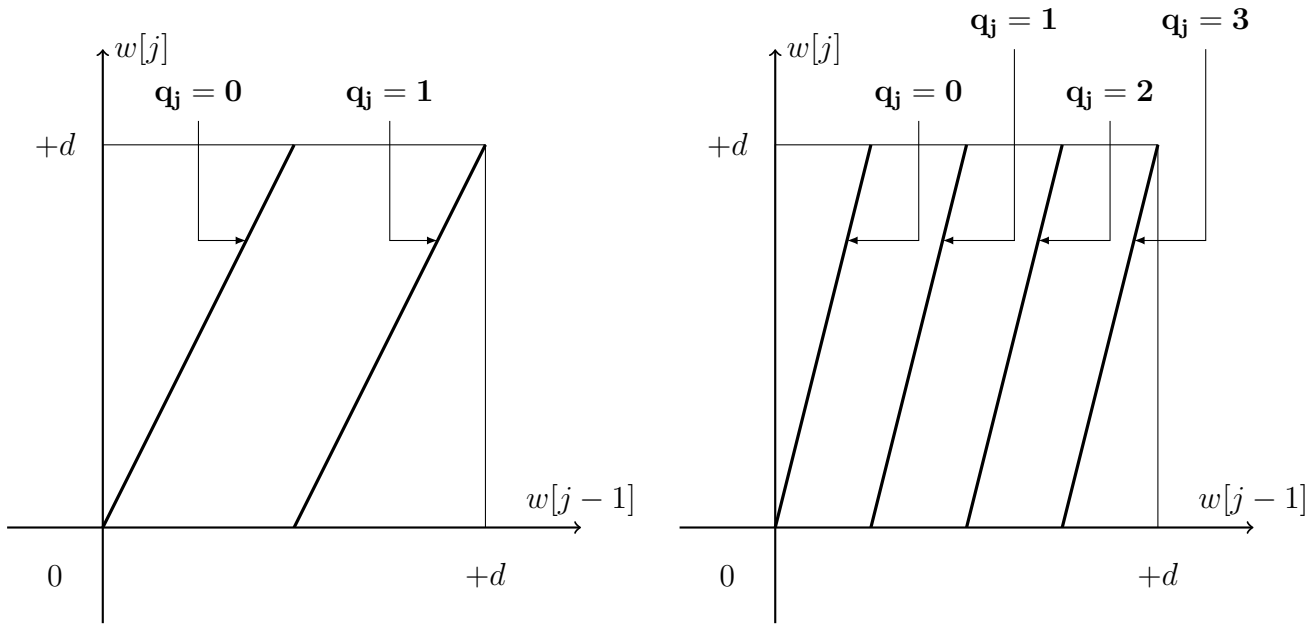

Figure 2.1 *Diagrammes de Robertson pour la division restaurante en base 2 (à gauche) et en base 4 (à droite).*

La fonction de sélection demande ici de comparer $2w[j-1]$ à d. Cette opération a un coût : elle demande un comparateur, c'est à dire une soustraction à propagation de retenue.

$$q_j = \text{Sel}(w[j-1], d) = \begin{cases} 1 & \text{si } (2w[j-1] - d) \geq 0, \\ 0 & \text{sinon.} \end{cases}$$

Algorithme 5 : *Division restaurante.*

$w[0] \leftarrow x$
pour $j = 1$ **à** n **faire**
  si $(2w[j-1] - d) \geq 0$ **alors**
    $w[j] \leftarrow 2w[j-1] - d$
    $q_j \leftarrow 1$
  sinon
    $w[j] \leftarrow 2w[j-1]$
    $q_j \leftarrow 0$

La fonction de sélection du chiffre peut s'illustrer par un *diagramme de Robertson* [48], qui montre pour chaque choix valide du prochain chiffre q_j du quotient les valeurs que prend $w[j]$ en fonction de $w[j-1]$. La figure 2.1 illustre les diagrammes de Robertson de la division restaurante pour les bases 2 et 4.

2.3.3 Division non restaurante

Il est possible d'améliorer la division restaurante en intégrant l'étape de restauration dans les itérations suivantes. L'algorithme résultant est appelé algorithme de *division non restaurante*. Il utilise l'ensemble de chiffres $\{-1, +1\}$ pour effectuer en même temps la mise à jour du reste partiel et la sélection du chiffre du quotient.

La fonction de sélection demande toujours une comparaison, celle de $w[j-1]$ et de 0. Cette opération n'est pas coûteuse lorsque le reste partiel est dans une représentation non redondante, puisqu'il suffit de tester son bit de signe. Mais si l'on veut utiliser un reste partiel redondant, afin d'accélérer le calcul de la récurrence $w[j] = rw[j-1] - dq_j$, le test du signe va de nouveau demander une addition à propagation de retenue. On a donc toujours au moins une addition/soustraction à propagation de retenue dans le chemin critique.

$$q_j = \text{Sel}(w[j-1], d) = \begin{cases} +1 & \text{si } w[j-1] \geq 0, \\ -1 & \text{si } w[j-1] < 0. \end{cases}$$

La division non restaurante permet d'effectuer la même quantité de calculs à chaque itération, ce qui équilibre les chemins critiques. La conversion du quotient à partir de la représentation 2–1 vers la base de numération choisie pour la sortie peut être faite par une soustraction ou par une conversion au vol [11, p.256], présentée plus loin.

Algorithme 6 : *Division non restaurante.*

```
w[0] ← x
pour j = 1 à n faire
    si w[j − 1] ≥ 0 alors
        w[j] ← 2w[j − 1] − d
        q_j ← +1
    sinon
        w[j] ← 2w[j − 1] + d
        q_j ← −1
```

La division non restaurante effectue n décalages (du simple routage en matériel) et n additions /soustractions, ce qui en fait un algorithme plus rapide que la division restaurante [51, p. 180]. La figure 2.2 illustre les diagrammes de Robertson de la division non restaurante pour les bases 2 et 4.

2.3.4 Division SRT

Pour accélérer les algorithmes de division restaurante et non restaurante, on peut augmenter la base de travail r. La fonction de sélection demande alors $r-1$ comparateurs, ce qui en pratique limite les implantations à la base 2 ou 4. Chaque comparaison demande en effet une soustraction à propagation de retenue entre le reste partiel et un multiple du diviseur, ce qui est coûteux à la fois en temps et en surface de circuit.

La division SRT [48, 50] accélère le calcul de chaque itération en facilitant le choix du nouveau chiffre q_j du quotient. Pour ce faire, l'algorithme utilise une représentation redondante du quotient. Cette redondance autorise une certaine « erreur » lors du choix de q_j qui peut être corrigée lors des itérations suivantes. On utilise cette latitude dans le choix de q_j pour simplifier la fonction de sélection : le choix d'un chiffre ne se fait plus par une comparaison exacte, mais en utilisant des *estimations* du reste partiel et du diviseur. En pratique, la comparaison des estimations peut s'implanter à l'aide d'une table (ou une structure à base de tables : PLA, LUT, ...) qui sera plus petite et plus rapide que des comparateurs.

La convergence du résultat partiel vers le quotient est plus compliquée à assurer. On sait que l'erreur finale sur le quotient doit être positive et strictement inférieure au poids du LSB du quotient :

$$0 \leq \varepsilon = \frac{x}{d} - q < r^{-n}.$$

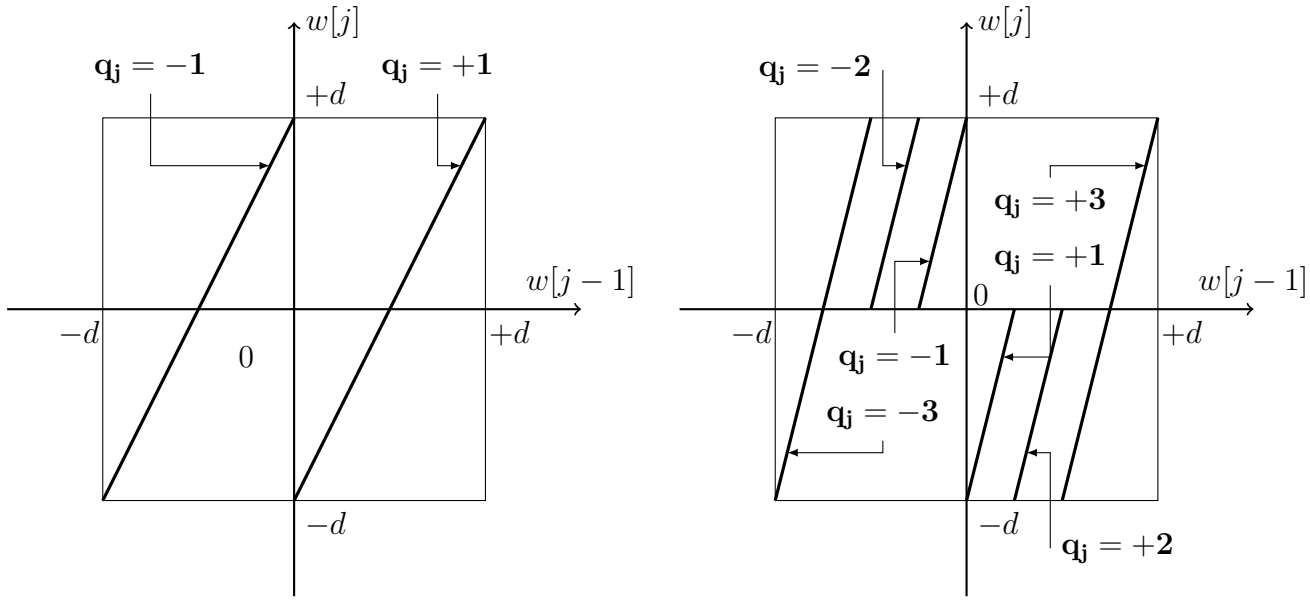

Figure 2.2 *Diagrammes de Robertson pour la division non restaurante en base 2 (à gauche) et en base 4 (à droite).*

La récurrence doit faire converger le résultat vers cette erreur. Si on note $\varepsilon[j]$ l'erreur à l'itération j, alors :

$$\varepsilon[j] = \frac{x}{d} - q[j].$$

Une étape finale corrige l'erreur $\varepsilon[n]$, qui peut être négative, pour donner un résultat avec arrondi fidèle. À l'itération j, il est possible de diminuer au plus l'erreur de αr^{-j}, en choisissant le chiffre le plus haut de la représentation utilisée. Pour pouvoir converger vers le bon résultat, on doit vérifier :

$$|\varepsilon[j]| \leq |\varepsilon[n]| + \sum_{i=j+1}^{n} \alpha r^{-i} = |\varepsilon[n]| + \frac{\alpha}{r-1}(r^{-j} - r^{-n}) = \rho r^{-j} + \left(|\varepsilon[n]| - \rho r^{-n}\right).$$

La convergence est assurée si $|\varepsilon[j]| < \rho r^{-j}$. Pour $d > 0$, cette borne sur l'erreur devient :

$$|x - dq[j]| < \rho d r^{-j}.$$

L'erreur et les bornes sur l'erreur décroissent en amplitude au cours des itérations. On obtient le reste partiel, dont l'encadrement conserve le même ordre de grandeur (il ne dépend pas de j), en multipliant la condition de convergence par r^j :

$$w[j] = r^j(x - dq[j]), \quad \text{borné par } |w[j]| < \rho d.$$

De la définition du résultat partiel, on déduit l'équation de récurrence :

$$-\rho d < w[j] = r w[j-1] - d q_j < +\rho d.$$

La valeur initiale du reste partiel doit permettre la convergence du quotient. Elle doit aussi vérifier $|w[0]| < \rho d$. En posant $q[0] = 0$, et si $\rho = 1$, on peut prendre $w[0] = x/2 < d$. Si $\rho < 1$, alors on doit avoir $w[0] = x/4 < \rho d$.

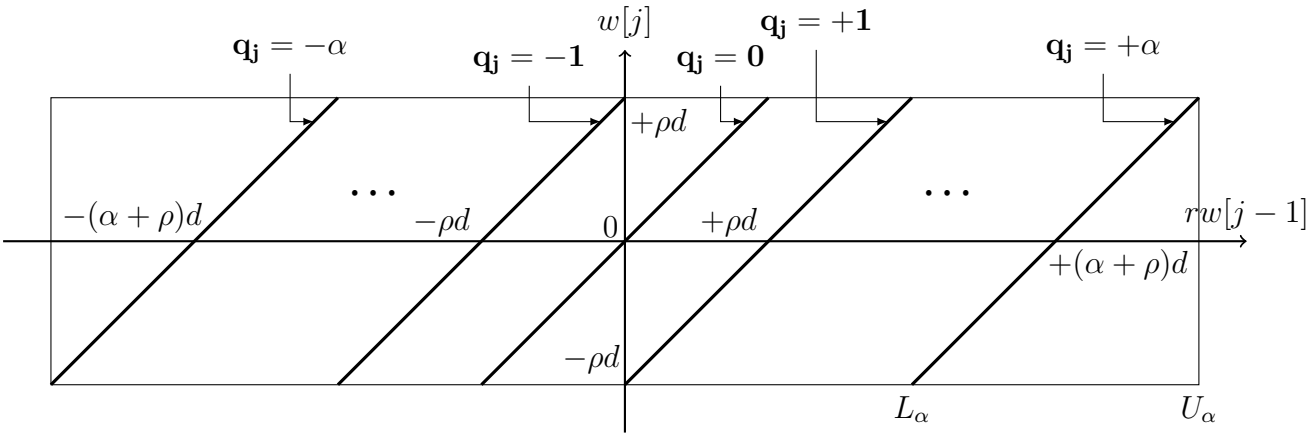

Figure 2.3 *Digramme de Robertson pour la division SRT en base r avec $\rho = 1$.*

Le choix de chaque chiffre q_j du résultat doit être fait de telle façon que la borne sur le reste partiel reste vérifiée. La fonction de sélection $q_j = \text{Sel}\,(w[j-1], d)$ est accélérée en utilisant, à la place des valeurs exactes du reste partiel et du diviseur pour les comparaisons, des estimations de ces valeurs notées $\hat{w}[j-1]$ et $\hat{d}$. Une estimation d'une valeur est faite en lisant seulement quelques uns des bits de poids forts de la valeur, et en négligeant les bits de rang inférieur. On a alors :

$$q_j = \text{Sel}\left(\hat{w}[j-1], \hat{d}\right).$$

Fonction de sélection du chiffre q_j

On cherche maintenant à déterminer une fonction de sélection valide, et qui permette un calcul rapide du nouveau chiffre q_j.

On définit pour chaque chiffre $k \in \{-\alpha, \ldots, \alpha\}$ de la représentation utilisée l'intervalle $]L_k(d), U_k(d)[$ des valeurs de $rw[j-1]$ pour lesquelles le choix du chiffre $q_j = k$ est correct. Par définition :

$$L_k(d) < rw[j-1] < U_k(d) \Rightarrow -\rho d < w[j] = rw[j-1] - kd < +\rho d.$$

On en déduit :

$$L_k(d) = (k-\rho)d, \quad U_k(d) = (k+\rho)d.$$

Sur le diagramme de Robertson correspondant, illustré par la figure 2.3, on constate que les zones de sélection pour deux chiffres successifs se recouvrent. C'est ce recouvrement qui permet d'effectuer la sélection à l'aide d'estimations du reste partiel et du diviseur, au lieu de devoir utiliser des comparateurs exacts.

Les intervalles de sélection $]U_k(d), L_k(d)[$ servent à caractériser la fonction de sélection. Une fonction de sélection peut être représentée à l'aide de l'ensemble de frontières $\{s_k\}$, où $k \in \{-\alpha, \ldots, \alpha\}$, vérifiant :

$$q_j = k \quad \text{si } s_k(d) \leq rw[j-1] \leq s_{k+1}(d) - \text{LSB}(w), \quad \text{avec } L_k(d) < s_k(d) < U_k(d).$$

La fonction de sélection doit aussi vérifier une condition de continuité : on doit toujours être capable de choisir un nouveau chiffre du quotient. On a donc pour $k > -\alpha$ la contrainte supplémentaire $s_k(d) - \text{LSB}(w) < U_{k-1}(d)$. On en déduit l'encadrement :

$$L_k(d) < s_k(d) < U_{k-1}(d) + \text{LSB}(w).$$

Dans l'algorithme de division restaurante par exemple, les frontières de sélection sont définies par $kd \leq s_k(d) < kd + \text{LSB}(w)$, ce qui impose $s_k = kd$. C'est la raison pour laquelle on a besoin d'un comparateur pour chaque chiffre $k \in \{0, \ldots, r-1\}$.

En division SRT, l'encadrement devient :

$$(k-\rho)d < s_k(d) < (k-1+\rho)d + \text{LSB}(w). \tag{2.1}$$

Cet encadrement a une largeur de $(2\rho - 1)d$. Plus il est large, et plus on va pouvoir choisir une frontière $s_k(d)$ facile à calculer. En pratique, on cherche à définir chaque frontière s_k à l'aide d'un ensemble de constantes $m_{k,v}$. Chaque constante $m_{k,v}$ sert à indiquer la valeur de $rw[j-1]$ à partir de laquelle le choix du chiffre $q_j = k$ est valide. Cette constante doit être valide pour toutes les valeurs du diviseur d dont l'estimation $\hat{d}$ vaut v. Elle doit vérifier :

$$\max_{\hat{d}=v}(L_k(d)) \leq m_{k,v} \leq \min_{\hat{d}=v}(U_{k-1}(d)) + \text{LSB}(w). \tag{2.2}$$

De la largeur de l'encadrement (2.1) va dépendre la simplicité matérielle de la fonction de sélection. Clairement, plus l'amplitude de d est faible et plus la frontière s_k implantée à l'aide de segments $m_{k,v}$ va être composée de segments petits. Il faudra alors des estimations plus précises du diviseur, ce qui entraînera une table de sélection plus coûteuse. Pour pallier ce problème, on normalise habituellement d, en restreignant sa valeur à l'intervalle $[\frac{1}{2}, 1[$. Si on note $\text{Est}(d)$ le nombre de bits de d lus pour obtenir son estimation $\hat{d}$ (puisque $d \in [\frac{1}{2}, 1[$, on lit les bits fractionnaires à partir du rang -2), alors on définit :

$$d_i = \frac{1}{2} + i \times 2^{\text{Est}(d)}.$$

D'autre part, le recouvrement entre les zones de choix pour les chiffres k et $k+1$ est nul pour $\rho = 1/2$ (on a alors besoin de comparateurs), et d'autant plus important que la redondance est grande, c'est à dire que le facteur ρ est proche de 1. Typiquement, la fonction de sélection est plus coûteuse en représentation 4–2 qu'en représentation 4–3. Une étude des différents compromis est faite dans la prochaine section.

La figure 2.4 illustre les bornes de la valeur m_{k,d_0} pour $\text{Est}(d) = 0$.

Estimation du diviseur et du reste partiel

En utilisant les estimations $\hat{d}$ et $\hat{w}[j-1]$ au lieu des valeurs exactes du diviseur et du reste partiel, on commet une erreur qui dépend de leur format de représentation. On note $\delta(\hat{y})$ l'intervalle de valeurs possibles de y lorsque son estimation vaut $\hat{y}$. En clair :

$$\text{si } y \text{ est estimé par } \hat{y}, \text{ alors } y \in \delta(\hat{y}).$$

L'estimation $\hat{y}$ d'une valeur y est faite en ne considérant que quelques uns de ses bits de poids forts. Lorsque y est écrite dans une notation redondante (par exemple la notation à retenue conservée), on doit effectuer une addition pour convertir l'estimation vers un format classique (comme du complément à 2). Pour plus de précision, on peut effectuer cette conversion sur des estimations utilisant g bits supplémentaires, dits bits de garde, que l'on additionne, puis prendre l'estimation finale. La table 2.1 présente l'erreur d'estimation pour plusieurs formats de représentation. Cette erreur doit être prise en compte dans le calcul de la fonction de sélection.

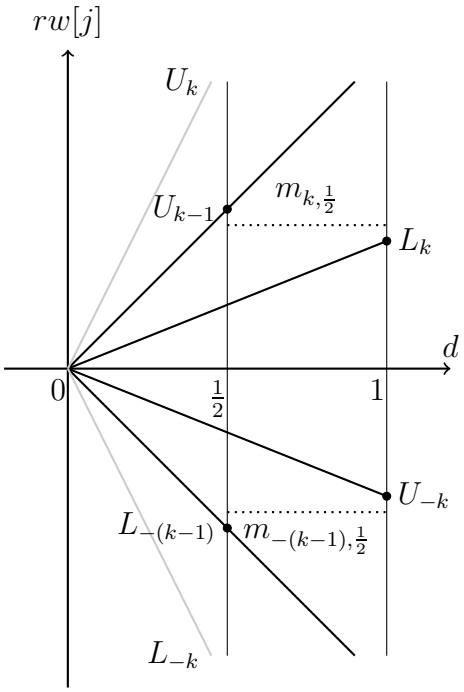

Figure 2.4 *Bornes de m_{k,d_0} ($d_0 = \frac{1}{2}$ et $k > 0$).*

Format de y	$\delta(\hat{y}) - \hat{y}$
Complément à 2	$[0, \text{LSB}(\hat{y}) - \text{LSB}(y)]$
Carry-Save	$[0, (1 + 2^{-g})\text{LSB}(\hat{y}) - 2\ \text{LSB}(y)]$
Borrow-Save	$[-2^{-g}\text{LSB}(\hat{y}) + \text{LSB}(y), \text{LSB}(\hat{y}) - \text{LSB}(y)]$

Table 2.1 *Valeur de $\delta(\hat{y})$ pour plusieurs formats de représentation.*

Les constantes m_{k,d_i} vont être comparées à l'estimation $\hat{w}[j-1]$ du reste partiel. On cherche donc à l'écrire avec une précision égale au poids $\text{LSB}(\hat{w})$. L'équation (2.2) pour une fonction de sélection exacte devient dans le cas d'estimations :

$$\left\lceil \max_{\hat{d}=v}(L_k) - \underline{\delta(\hat{w})} \right\rceil_{\text{LSB}(\hat{w})} \leq m_{k,v} \leq \left\lfloor \min_{\hat{d}=v}(U_{k-1}) - \overline{\delta(\hat{w})} \right\rfloor_{\text{LSB}(\hat{w})} + \text{LSB}(\hat{w}). \tag{2.3}$$

où $\lfloor x \rfloor_{\text{LSB}(\hat{w})}$ (respectivement $\lceil x \rceil_{\text{LSB}(\hat{w})}$) est x arrondi au multiple de $\text{LSB}(\hat{w})$ inférieur (respectivement supérieur). $\underline{\delta(\hat{w})}$ (respectivement $\overline{\delta(\hat{w})}$) est la borne inférieure (respectivement supérieure) de l'intervalle $\delta(\hat{w})$.

On remarque que le terme $\text{LSB}(w)$ qui vient de la condition de continuité dans l'équation (2.2) est devenu $\text{LSB}(\hat{w})$ dans l'équation (2.3). En effet, il suffit d'avoir un chiffre valide par couple d'estimations $(\hat{w}[j-1], \hat{d})$ possible. Si on a négligé $\text{LSB}(w)$, qui est petit, au cours de la démonstration, on perd plus tard $\text{LSB}(\hat{w})$ qui lui est grand (on estime le reste partiel sur aussi peu de bits que possible). Ce terme facilite beaucoup la création d'une fonction de sélection valide, car il élargit considérablement les plages de choix pour les m_{k,d_i}.

Le but est d'obtenir une fonction de sélection économique. On veut donc pouvoir utiliser des estimations grossières du diviseur et du reste partiel, c'est à dire qui reposent sur peu de bits. Il est possible d'établir des équations analytiques qui donnent le nombre minimum de bits pour ces estimations [43].

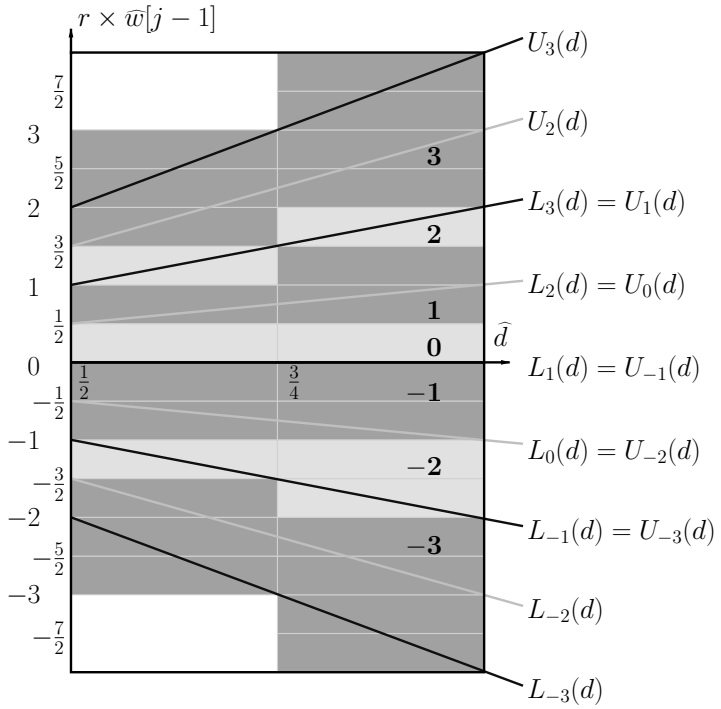

Figure 2.5 *Diagramme P-D d'une division SRT en représentation 4–3.*

Le comportement d'une fonction de sélection s'illustre généralement à l'aide d'un *diagramme P-D*, qui montre les zones de choix de chaque chiffre $q_j = k$ en fonction des valeurs respectives des estimations $\hat{d}$ du diviseur et $r\hat{w}[j-1]$ du reste partiel. Le diagramme P-D d'une fonction de sélection pour la division en représentation 4–3, pour un reste partiel en complément à 2, est illustré par la figure 2.5. Chaque rectangle qui apparaît sur ce diagramme représente une combinaison d'estimations $(r\hat{w}[j-1], \hat{d})$, et donne un chiffre q_j du quotient valide qui y correspond. Par exemple, si l'estimation $\hat{d}$ vaut $\frac{3}{4}$, et si $r \times \hat{w}[j-1]$ vaut $\frac{3}{2}$, on peut lire que $q_j = 2$ convient. La complexité mathématique et matérielle de la fonction de sélection peut se caractériser *à priori* par le nombre de « cases » qui sont ainsi spécifiées. Chaque case se traduit en matériel par une entrée d'une table, avec en sortie le chiffre q_j donné pour cette case.

Algorithme 7 : *Division SRT.*

si $\rho = 1$ **alors**
  $w[0] \leftarrow x/2$
sinon
  $w[0] \leftarrow x/4$
pour $j = 1$ **à** n **faire**
  $q_j = \text{Sel}\left(\hat{w}[j-1], \hat{d}\right)$
  $w[j] = rw[j-1] - dq_j$

Exemple 2.3 *On considère un diviseur SRT qui prend un dividende en complément à 2 sur 16 bits, un diviseur non signé sur 8 bits et retourne un quotient en complément à 2 sur 8*

bits. Leurs tailles et formats respectifs permettent à x, d et q d'être tous considérés comme entiers ($\sigma = 1$). Ce diviseur travaille en représentation 4–3.
Dans ces conditions, il est possible d'obtenir une fonction de sélection qui prend un argument un seul bit de d (de rang -2), et quatre bits de w.

$$x = (30000)_{10} = (111010100110000)_2,$$

$$d = (250)_{10} = (11111010)_2,$$

reste partiel initial : $w[0] = x/2$.

$w[0] = (0)0111010100110000 \quad (\hat{d}, \hat{w}[0]) = (1, 0011) \quad q_0 = 2$

$w[1] = (1)1111000001100000 \quad (\hat{d}, \hat{w}[1]) = (1, 1111) \quad q_1 = -1$

$w[2] = (0)0111110100000000 \quad (\hat{d}, \hat{w}[2]) = (1, 0011) \quad q_2 = 2$

$w[3] = (0)0000000000000000 \quad (\hat{d}, \hat{w}[3]) = (1, 0000) \quad q_3 = 0$

Le quotient calculé est $q = (2\overline{1}20)_4 = (120)_{10}$. La simulation logique correspondante est illustrée dans la figure 2.6.

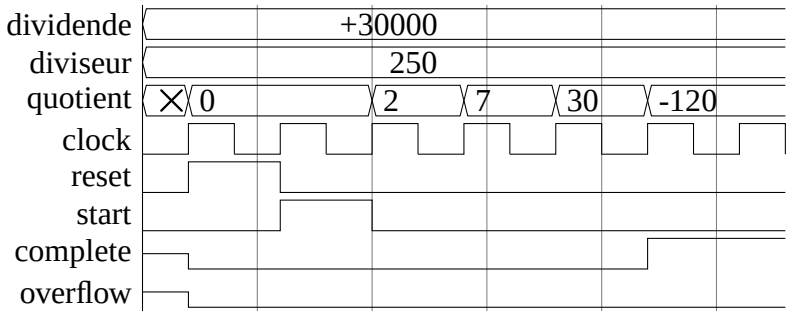

Figure 2.6 *Division de 30000 par 250.*

2.3.5 Conversion du quotient

Le quotient obtenu par les divisions non restaurante et SRT est en notation redondante r–α. Il doit être reconverti en notation simple de position avant d'être réutilisé ou interprété. La façon la plus naturelle de le reconvertir en complément à 2 est de le séparer en deux vecteurs, l'un contenant les chiffres positifs du quotient, l'autre les valeurs absolues de ses chiffres négatifs, puis de soustraire ces vecteurs entre eux. Un telle reconversion introduit un délai dû à l'addition. Un méthode de conversion au vol existe [41] qui permet d'effectuer la conversion au fur et à mesure que les chiffres du quotient sont produits. Cette conversion a aussi été étendue pour permettre l'arrondi du quotient [42]. Les détails peuvent être trouvés dans [51].

Soit $q[j]$ le quotient partiel à l'itération j, déjà converti en numération simple de position. Alors :

$$q[j+1] = q[j] + q_{j+1} r^{-(j+1)}.$$

Puisque q_{j+1} peut être négatif, on distingue deux cas :

$$q[j+1] = \begin{cases} q[j] + q_{j+1} r^{-(j+1)} & \text{si } q_{j+1} \geq 0, \\ q[j] - r^{-j} + (r - |q_{j+1}|) r^{-(j+1)} & \text{si } q_{j+1} < 0. \end{cases}$$

Dans le cas $q_{j+1} \geq 0$, l'addition consiste en une simple concaténation, car on sait que $q_{j+1} < r$. Par contre, la soustraction $q[j] - r^{-j}$ est coûteuse, car elle demande la propagation d'une retenue. Pour la simplifier, on introduit une variable $q^-[j]$ sur laquelle on applique le même raisonnement. Elle vaut :

$$q^-[j] = q[j] - r^{-j}.$$

On note la concaténation du mot $q[j]$ et du chiffre c sous la forme $q[j]\&c$. Le système résultant ne contient plus d'additions ou soustractions à propagation. Il est défini par :

$$q[j+1] = \begin{cases} q[j] \ \& \ q_{j+1} & \text{si } q_{j+1} \geq 0, \\ q^-[j] \ \& \ (r - |q_{j+1}|) & \text{si } q_{j+1} < 0. \end{cases}$$

$$q^-[j+1] = \begin{cases} q[j] \ \& \ (q_{j+1} - 1) & \text{si } q_{j+1} > 0, \\ q^-[j] \ \& \ (r - 1 - |q_{j+1}|) & \text{si } q_{j+1} < 0. \end{cases}$$

L'opérateur réalisant la conversion au vol est simple à implanter. Il demande deux registres et deux multiplexeurs de la taille du quotient final, plus un peu de logique pour le calcul du chiffre ajouté et des signaux de contrôle.

2.4 Le programme `Divgen`, un générateur de diviseurs

`Divgen` est un générateur de diviseurs en virgule fixe. Étant donné un ensemble de paramètres et d'options, il génère automatiquement une description VHDL optimisée du diviseur correspondant. `Divgen` est un projet de recherche en cours, qui a été présenté lors de la conférence internationale *Advanced Signal Processing Algorithms, Architectures and Implementations XV* à San Diego en 2005 [5]. Il est mis à disposition sous la licence GPL avec les possibilités décrites par la suite. Ce travail sera complété par plus d'options et d'algorithmes, et étendu à d'autres opérateurs arithmétiques.

Les tailles de diviseurs rendues accessibles par `Divgen` vont de 4 à 64 bits. En dehors de cette fourchette, des méthodes et des architectures plus efficaces existent [11].

2.4.1 Architecture générale des diviseurs

Le programme génère un ficher résultat qui contient une description VHDL des composants utiles et une architecture générale intitulée "Divider" (illustrée dans les figures 2.7 et 2.8. La liste des ports et leur fonction est décrite dans la table 2.2).

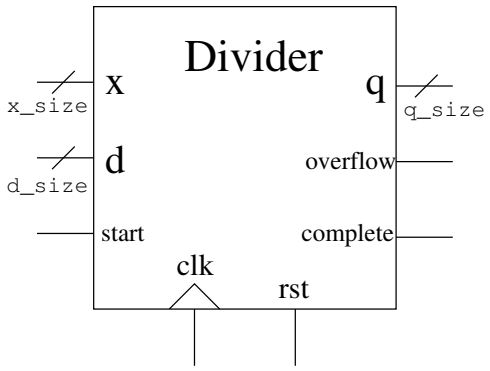

Figure 2.7 *Architecture générale du diviseur généré.*

```
entity Divider is
  port (
    dividend : in std_logic_vector(31 downto 0);
    divisor : in std_logic_vector(15 downto 0);
    clock : in std_logic;
    reset : in std_logic;
    start : in std_logic;
    quotient : out std_logic_vector(15 downto 0);
    overflow : out std_logic;
    complete : out std_logic );
end entity;

architecture arch of Divider is
      [...]
end architecture;
```

Figure 2.8 *En-tête VHDL du diviseur.*

Nom du port	Entrée/Sortie	Taille	Rôle
x	E	*entière*	entrée parallèle du dividende
d	E	*entière*	entrée parallèle du diviseur
start	E	1	démarre le calcul
clock	E	1	horloge
reset	E	1	réinitialise l'opérateur
q	S	*entière*	sortie parallèle du quotient
overflow	S	1	signale un dépassement
complete	S	1	complétion du calcul

Table 2.2 *Liste de ports du diviseur.*

2.4.2 Options générales

Les valeurs x, d et q sont des entiers ou des nombres représentés en virgule fixe. Les valeurs en virgule flottante ne sont pas prises en compte dans cette version. Chaque valeur est caractérisée par sa taille en bits et sa représentation (non signée, signe et valeur absolue ou complément à 2).

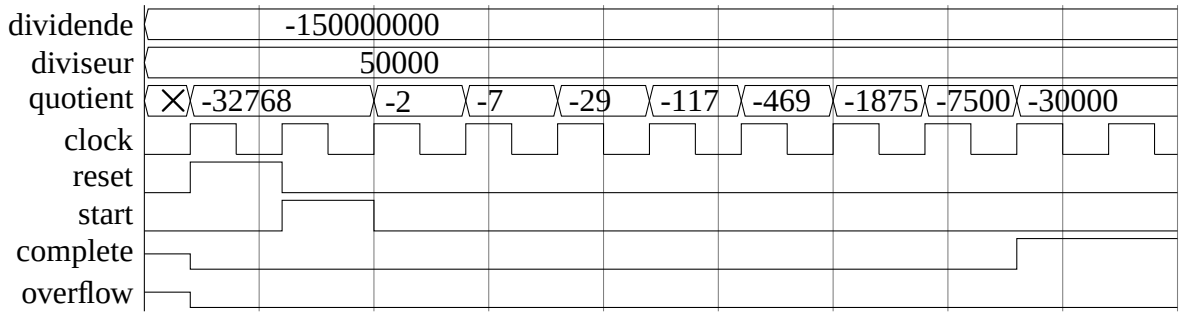

Figure 2.9 *Forme d'onde pour une division 32 bits en complément à 2 par 16 bits non signés. La sortie est en complément à 2 sur 16 bits, et l'algorithme travaille dans la représentation 4–3.*

L'utilisation de la notation à retenue conservée pour le reste partiel permet d'accélérer l'étape de récurrence. Les algorithmes supportés ont été testés pour des tailles d'opérandes allant de 4 à 64 bits.

Trois algorithmes sont jusqu'ici disponibles : restaurant, non restaurant et SRT. Tous trois sont des algorithmes à récurrence de chiffres, qui produisent un nombre fixe de chiffres du quotient à chaque itération [51, 11, 46] (un chiffre du quotient peut représenter plusieurs bits suivant la base de numération), en partant des chiffres de poids fort. Ces algorithmes utilisent des décalages, additions et un peu de logique supplémentaire, par exemple des petites tables pour la fonction de sélection. Les choix de la base de travail r et de l'amplitude de l'ensemble redondant de chiffres utilisés $\{-\alpha, \ldots, +\alpha\}$ (pour la division SRT) ont une influence forte sur la latence de l'opérateur [11]. De manière générale, accroître la base de travail diminue le nombre d'itérations nécessaires pour atteindre une précision équivalente du quotient. Par contre, lorsque la base augmente, les itérations ont tendance à devenir plus compliquées, et la latence globale de la division n'est pas forcément diminuée comme on s'y attend. À bases de travail r égales, une amplitude de α plus grande permet généralement de simplifier la fonction de sélection (en augmentant ρ, donc le recouvrement des zones de sélection). Mais on doit alors générer plus de multiples du diviseur, ce qui ajoute au coût matériel de l'opérateur.

La différence majeure entre les algorithmes proposés est le fait que les divisions restaurantes et non restaurantes peuvent travailler sur n'importe quelles entrées, alors que la division SRT demande la normalisation du diviseur, c'est à dire que l'on doit vérifier :

$$2^{\texttt{d\_size}-1-s}\mathrm{LSB}(d) \leq |d| \leq 2^{\texttt{d\_size}-s}\mathrm{LSB}(d),$$

où `d_size` représente la taille du diviseur, et $s = 0$ si d est non signé, $s = 1$ sinon.

Les paramètres communs à tous les algorithmes sont les suivants :

- `dividend_representation` [unsigned | sign-magnitude | 2s_complement]
  spécifie la représentation du dividende.
- `divisor_representation` [unsigned | sign-magnitude | 2s_complement]
  spécifie la représentation du diviseur.
- `quotient_representation` [unsigned | sign-magnitude | 2s_complement]
  spécifie la représentation du quotient.
- `dividend_size` [*entier*]
  spécifie la taille du dividende en bits.
- `divisor_size` [*entier*]
  spécifie la taille du diviseur en bits.

- `quotient_size` [*entier*]
  spécifie la taille du quotient en bits.
- `component_algorithm` [restoring | nonrestoring | SRT]
  choisit l'algorithme de division.

Ci-dessous est présenté un exemple de fichier de configuration, qui demande la génération d'un diviseur 32 bits non signé fonctionnant à l'aide de l'algorithme restaurant.

```
x_representation unsigned
d_representation unsigned
q_representation unsigned
x_size 32
d_size 32
q_size 32
algorithm restoring
```

2.4.3 Options spécifiques à la division SRT

Les diviseurs SRT peuvent être générés avec plusieurs options :

- `quotient_radix` [*puissance_de_2*]
  la base de travail r.
- `quotient_max_digit` [*entier*]
  l'amplitude de l'ensemble de chiffres du quotient α.
- `partial_remainder_representation` [2s_complement | carry-save]
  la représentation redondante du reste partiel. On utilise habituellement une notation en complément à 2 sur FPGA lorsque les lignes de *fast-carry* sont disponibles, et une notation à retenue conservée sur circuits ASIC.
- `step_adder` [RCA | CSA]
  le type d'additions utilisées pour le calcul de la récurrence. Il s'agit habituellement d'additionneurs à propagation de la retenue (RCA) sur FPGA, et d'additionneurs parallèles carry-save sur ASIC.
- $\#$`guard_bits` [*entier*]
  bits supplémentaires utilisés pour le calcul d'une estimation non redondante du reste partiel quand on travaille avec un reste partiel en notation à retenue conservée.
- `SRT_table_folding` [yes | no]
  repli de la table de sélection sur elle-même, comme proposé dans [47]. Ceci permet de diviser son coût matériel par 2, en ajoutant simplement quelques portes XOR.
- `gray_encoding` [yes | no]
  application d'un code de Gray sur la table de sélection, afin de préserver la redondance autant que possible. Cette redondance peut permettre certaines simplifications logiques lors de l'implantation (le choix final d'un chiffre est laissé au synthétiseur).

Chaque option a une influence complexe sur la fonction de sélection et sur les performances de l'opérateur (voir [47] par exemple). Nous montrons plus loin quelques situations où elles s'avèrent utiles.

On illustre le comportement de la fonction de sélection à l'aide de diagrammes P-D, générés dans un fichier xfig par `Divgen` en plus de la description VHDL de l'opérateur. Les bornes théoriques L_k et U_k sont illustrées par les lignes continues d'origine $(0, 0)$. Puisque le diviseur et le reste partiel sont seulement estimés, la valeur Sel$(\hat{d}, \hat{w}[j-1])$ sera toujours la même pour

une plage de d et $w[j-1]$ correspondant à la même estimation. Ces plages sont illustrées par des rectangles, qui à leur tour seront implantés sous la forme d'entrées $(\hat{d}, \hat{w}[j-1])$ d'une table dont la sortie est q_j. La taille de la table dépend directement de la précision des estimations : un bit supplémentaire dans une des estimations double le nombre d'entrées.

Dans la suite, on caractérise *à priori* la complexité d'une fonction de sélection par le nombre de ses termes (le nombre de boites du diagramme P-D). Il s'agit en pratique d'une assez bonne estimation de son coût matériel.

`Divgen`, pour un ensemble de paramètres, calcule automatiquement une table de sélection valide et de taille minimale. Le fonctionnement de l'algorithme utilisé est détaillé dans la section 2.4.4.

Impact de r et α

Lorsque l'on augmente la valeur de la base de travail r, on diminue le nombre d'itérations nécessaires. Mais ceci veut aussi dire que l'on va devoir calculer plus de multiples $d \times q_j$ du diviseur d. Il y a plus de valeurs possibles de q_j, car on sait que $\alpha \geq \frac{r}{2}$. La fonction de sélection va aussi être plus compliquée, car chaque itération doit annuler $\log_2 r$ chiffres du reste partiel.

À r donné, augmenter la valeur de α veut dire que l'on doit générer plus de multiples du diviseur. Mais ceci augmente aussi la valeur de $\rho = \frac{\alpha}{r-1}$, donc le recouvrement des zones de sélection pour chaque chiffre. La fonction de sélection en est simplifiée, ce qui diminue le coût matériel de son implantation.

La figure 2.10 donne une idée générale de la façon dont sont calculés les multiples du diviseur. On a $m_0 \in \{-2, -1, 0, 1, 2\}$, $m_1 \in \{-4, 0, 4\}$ et $m_2 \in \{-8, 0, 8\}$.

La figure 2.11 présente les diagrammes P-D des fonctions de sélection pour plusieurs représentations du quotient. La table de sélection (a) comporte 272 termes, (b) 28 termes et (c) en comporte 400.

On constate que pour un r donné, un α plus grand *peut* compliquer la génération des multiples. Plus de multiples à générer ne veut pas forcément dire un opérateur plus complexe car la même structure de génération des multiples peut permettre le calcul de plusieurs ensembles de multiples, comme illustré dans la table 2.3. Par contre, un α plus grand simplifie toujours la fonction de sélection, parfois de façon impressionnante : on constate un facteur 10 dans le nombre de termes entre les représentations 4–2 et 4–3.

α	2		3	4 à 6		7 à 10		11 à 14		15 à 26		27 à 42		...
								8		16		32	16	
Structure			2	4		8	4	4		8	4	8	4	
	2	1	1	2	1	2	1	2	1	2	1	2	1	

Table 2.3 *Structures utilisées pour la génération des multiples de d.*

Redondance du reste partiel

L'utilisation d'un reste partiel écrit dans un système de numération à retenue conservée ou en borrow-save permet d'accélérer l'étape de récurrence $w[j] = rw[j-1] - q_j d$. La latence de la soustraction devient indépendante de la largeur des opérandes, et est en général bien plus faible qu'avec un reste partiel en complément à 2.

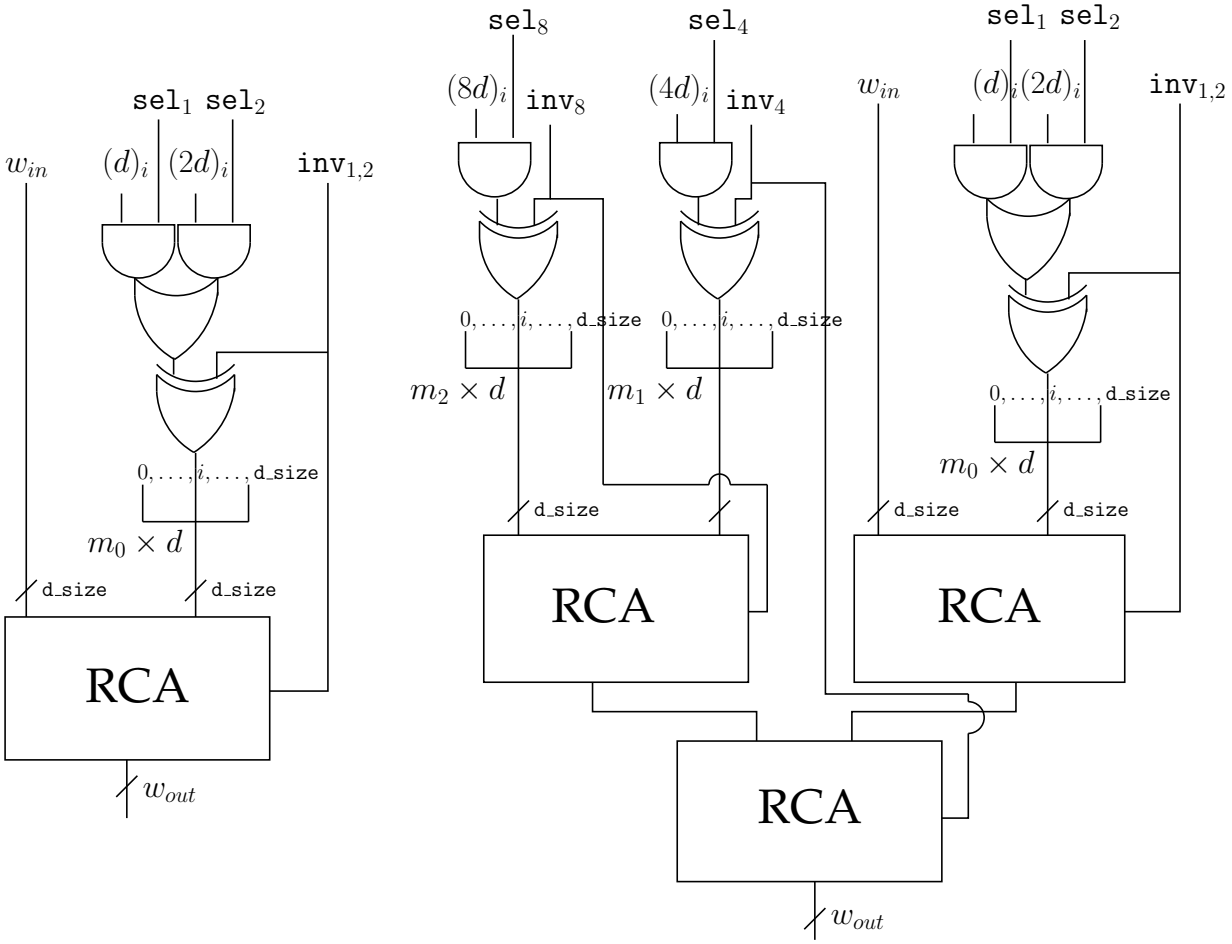

Figure 2.10 *Structures matérielles calculant la récurrence* $w[j] = rw[j-1] + q_j d$ *pour* $\alpha = 2$ *(à gauche) et* $10 < \alpha \leq 14$ *(à droite).*

Plutôt que d'utiliser une estimation redondante de w en entrée de la table, on le reconvertit dans une numération simple de position, ce qui diminue le nombre de bits en entrée de la table. On a vu (table 2.1) qu'une estimation en notation à retenue conservée sera moins précise qu'une estimation utilisant le même nombre de bits en complément à 2. Une précision moindre dans l'estimation fait diminuer la largeur des plages de sélection utilisables, donc augmente le nombre de bits nécessaires pour les estimations, et par là même la taille de la table de sélection. L'ajout de bits de garde pour augmenter la précision de l'estimation permet souvent de compenser l'effet d'un passage vers une représentation redondante et de simplifier la fonction de sélection [47] au prix de quelques cellules d'addition supplémentaires.

> **Exemple 2.4** *Pour une division SRT 8–6 avec un reste partiel en complément à 2, la fonction de sélection est définie par 686 termes. Si on passe à un reste partiel en notation à retenue conservée, et sans bits de garde (*$g = 0$*) pour l'addition précédent l'estimation, alors la table est composée de 2702 entrées.* $g = 1$ *conduit à 1358 termes, et pour* $g = 4$*, on retombe à une fonction de sélection en 686 cas.*
> *À titre de comparaison, la figure 2.12 montre les tables de sélection en SRT 4–3 pour un reste partiel en complément à 2 et en notation à retenue conservée pour* $g = 0$*.*

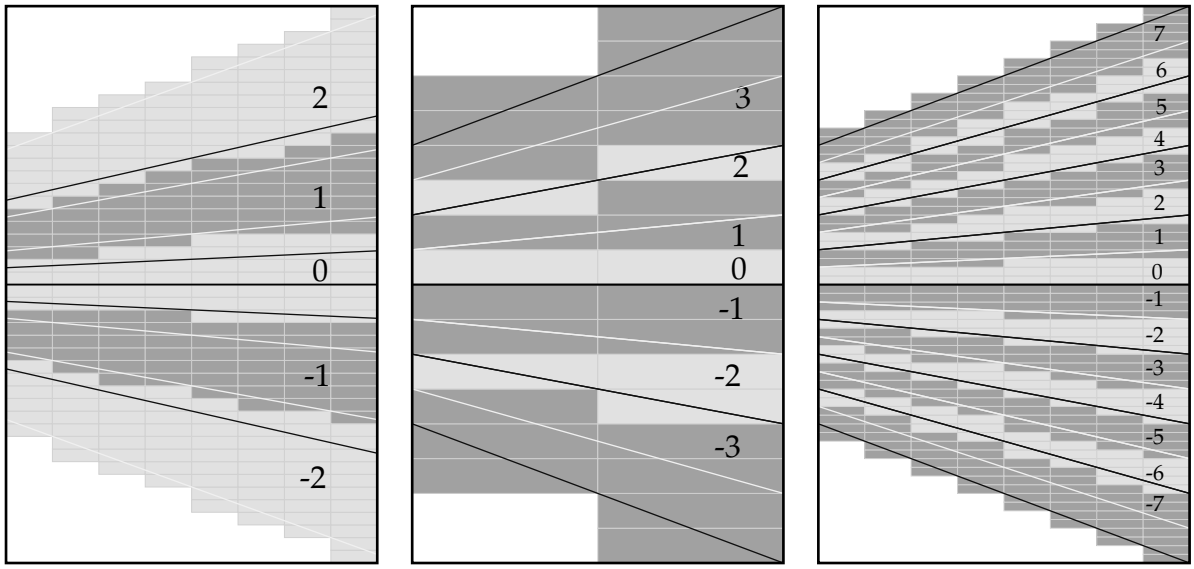

Figure 2.11 *Diagrammes P-D pour différentes représentations r–α : 4–2 (a), 4–3 (b) et 8–7 (c).*

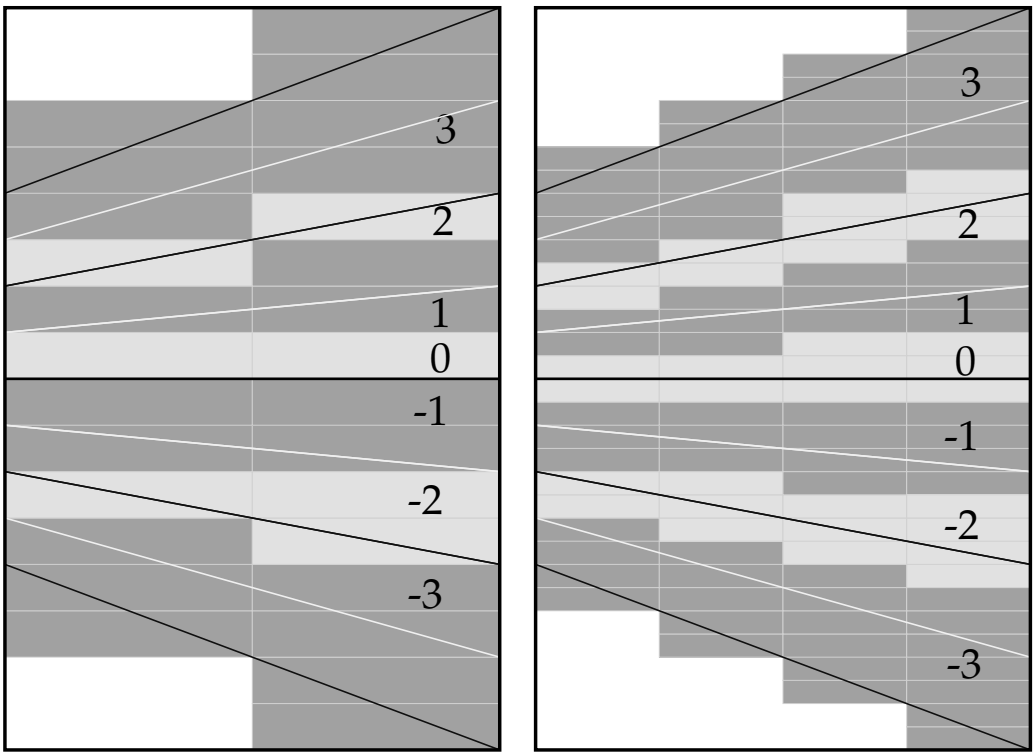

Figure 2.12 *Diagrammes P-D pour différentes la division SRT 4–3 avec un reste partiel en complément à 2 (à gauche), et en notation à retenue conservée sans bits de garde pour l'estimation (à droite).*

Repli de la table

Du fait de la symétrie de l'ensemble de chiffres $\{-\alpha, \ldots, +\alpha\}$ de la représentation utilisée, la table de sélection est presque symétrique par rapport à l'axe $\hat{w}[j-1] = 0$. *Presque* seulement, car l'erreur d'estimation $\delta(\hat{w})$ n'est en général pas symétrique (l'exception étant l'estimation d'un reste partiel en borrow-save sans bits de garde). Il est possible de replier la table sur l'axe $\hat{w}[j-1] = 0$ pour essayer de gagner un bit (le bit de signe) sur l'estimation $\hat{w}[j-1]$ du reste partiel. On multiplie simplement le chiffre en sortie de la table par le signe de $w[j-1]$. La fonction de sélection n'étant pas parfaitement symétrique, ce repli peut parfois demander une meilleur estimation de d et $w[j-1]$, qui va contrebalancer le gain réalisé voire mener à une augmentation globale du coût matériel. En pratique, on ne replie pas exactement la table. En effet, l'expression $|\hat{w}[j-1]|$ demande de calculer l'opposé de l'estimation $\hat{w}[j-1]$ lorsque celle-ci est négative. En complément à 2, le calcul de l'opposé d'une valeur v consiste à inverser ses bits puis à lui ajouter $\text{LSB}(v)$, ce qui demande une addition à propagation. On va plutôt se contenter d'inverser les bits de l'estimation $\hat{w}[j-1]$ lorsqu'elle est négative, en réalisant un XOR des bits de $\hat{w}[j-1]$ avec son bit de signe, et sans ajouter le $\text{LSB}(\hat{w})$. Cette valeur est moins chère à calculer que la véritable valeur absolue, mais elle introduit de plus une erreur qui compense (pour le complément à 2) celle due aux estimations. Au final, on paie moins de matériel, et le repli ne fait plus diminuer la largeur des plages de redondance.

Exemple 2.5 *Le table de sélection pour une division SRT en représentation 4–3 (respectivement 4–2) avec un reste partiel en complément à 2 est définie par 28 termes (respectivement 272). Après un replis exact à l'aide de $|\hat{w}[j-1]|$, elle contient au total 52 termes (respectivement 269). En approchant la valeur absolue à l'aide de portes* XOR*, la table ne contient plus que 14 termes (respectivement 136). La figure 2.13 illustre les tables de sélection de la division SRT repliées pour la représentation 4–3.*

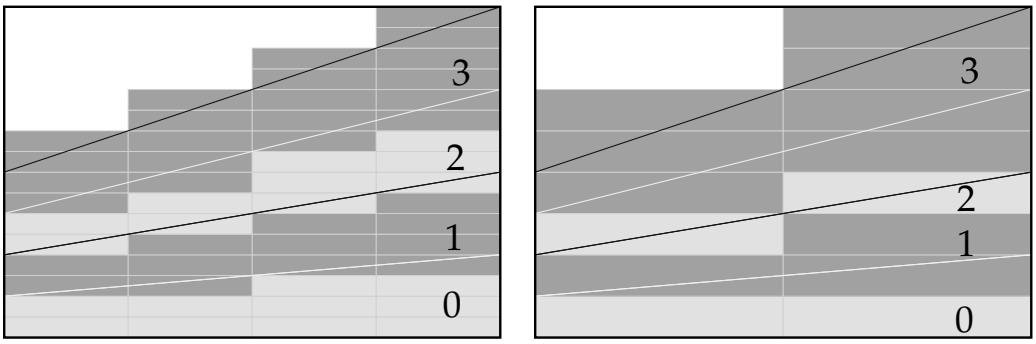

Figure 2.13 *Diagrammes P-D après repli pour la division SRT en utilisant la vraie valeur absolue $|\hat{w}[j-1]|$ comme estimation (à gauche), et la valeur approchée $\hat{w}[j-1]$ XOR signe$(\hat{w}[j-1])$ (à droite).*

Codage de Gray

Le recouvrement des plages de sélection fait qu'il subsiste des entrées du tableau pour lesquelles deux valeurs consécutives sont valides pour q_j. On perd habituellement cette redondance en choi-

sissant arbitrairement une des deux valeurs. `Divgen` permet via une option de préserver cette redondance jusqu'au niveau du VHDL, en représentant à l'aide d'un code de Gray les deux chiffres possibles (qui ne diffèrent que d'une position, où on écrit un « don't care »). Le synthétiseur peut ensuite choisir d'implanter l'une ou l'autre possibilité, afin de minimiser le coût matériel de la table. Nous envisageons aussi de tirer parti de cette possibilité pour des applications où la consommation électrique est critique : si possible, on va choisir le chiffre qui entraînera le moins de transitions dans l'arbre de génération des multiples. Si le codage de Gray n'est pas utilisé, le générateur choisit le chiffre du quotient minimal. La figure 2.14 illustre la fonction de sélection pour une division SRT 4–3 qui préserve la redondance.

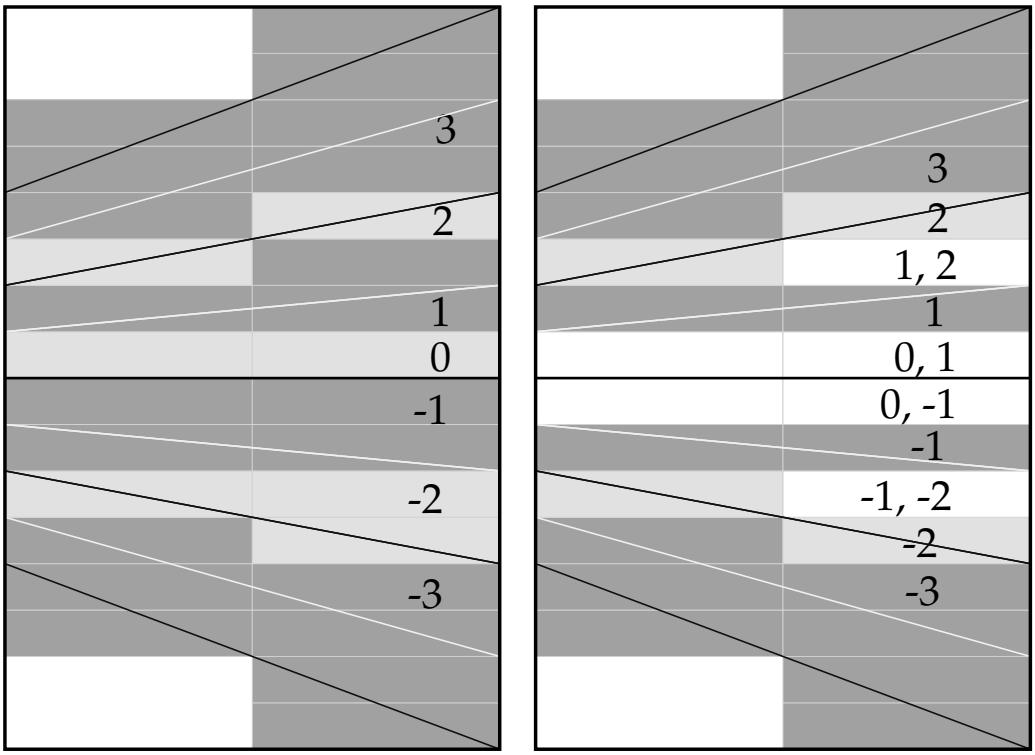

Figure 2.14 *Diagrammes P-D pour la division SRT en représentation 4–3, en choisissant le chiffre minimal (à gauche), et en préservant la redondance par codage de Gray (à droite).*

2.4.4 Détails techniques

Le programme `Divgen`

`Divgen` est publié sous licence GPL. Ce programme consiste en un ensemble de classes C++ utilisées pour créer une image mémoire d'un circuit qui peut être traduite en une description VHDL. Tous les composants sont dérivés de la classe `element`, qui implante les attributs et méthodes de base des composants. Chaque objet de la classe `element` peut contenir de nombreux autres éléments interconnectés et avec les ports de l'élément global, avec la possibilité de décrire des comportements récursifs et conditionnels. Un composant est crée en lui spécifiant des ports et des valeurs pour ses paramètres génériques. Les fils et bus sont représentés par la classe `signal`. Des opérateurs ont étés surchargés pour permettre une manipulation simple et proche de celle du VHDL (affectation, sélection, concaténation, ...).

Lors d'une exécution de `Divgen`, le fichier de configuration est lu, et le composant spécifié est crée en mémoire, avec tous ses sous-composants. Cette structure est ensuite parcourue et traduite en VHDL. Les assignations de signaux et créations de composants sont automatiquement traduites dans la grammaire, mais le comportement des composants basiques est spécifié dans le programme.

La représentation en mémoire du composant est générique. Pour l'instant, notre programme ne permet qu'une traduction vers le VHDL. À terme, nous voulons ajouter d'autres grammaires, comme Verilog ou SystemC. Nous voulons aussi être capables de générer automatiquement des programmes C ou C++ se comportant exactement comme le composant. Ceci permettra une validation fiable et rapide des opérateurs générés.

Les classes de composants utilisées par `Divgen` sont les suivantes :

```
element
   |- logic
   |     |- and2
   |     |- or2
   |     |- ...
   |- sequential
   |     |- register
   |- arithmetic
   |     |- RCA
   |     |- divider
   |     |     |- restoring
   |     |     |- nonrestoring
   |     |     |- SRT
```

Génération de la table SRT

Étant donné une représentation r–α, `Divgen` génère une table SRT optimisée qui lit en entrée Est(w) bits de $w[j-1]$ et Est(d) bits de d, et retourne un chiffre q_j tel que $w[j] = rw[j-1] - dq_j$ satisfait la condition de convergence.

On vise une taille d'entrée de la table Est(w) + Est(d) minimale. Le programme recherche la meilleure table en incrémentant la taille totale de l'entrée jusqu'à ce qu'une table valide soit atteinte. Il s'agit d'un simple parcours dans $\mathbb{N}^2$, avec une limite sur la taille d'entrée maximale. La condition de validité utilisée est celle décrite dans la présentation de la division SRT (section 2.3.4).

2.4.5 Résultats

On présente ici quelques résultats d'implantation pour les diviseurs générés par notre programme. Les implantations sont réalisés sur des FPGA Virtex-E 300 de Xilinx.

Les performances globales des diviseurs sont testées en fonction de la taille du quotient pour plusieurs algorithmes et options. La taille du dividende (en complément à 2) est le double de celle du diviseur (non signé) et du quotient (en complément à 2). La figure 2.15 donne la période d'horloge, la figure 2.16 le temps de calcul total, la figure 2.17 la surface des opérateurs, et la figure 2.18 illustre le produit du temps de calcul par la surface, le tout en fonction de la taille du quotient pour plusieurs algorithmes proposés.

Les tables 2.4 et 2.5 montrent l'influence des différents efforts de synthèse et en placement-routage, dans le cas d'un diviseur 32 bits par 16 bits. La table 2.6 illustre l'impact des paramètres sur la performance et le coût matériel des diviseurs.

Effort de synthèse	Normal			Fort		
Effort P&R	Standard	Moyen	Haut	Standard	Moyen	Haut
Surface (tranches)	193	193	193	193	193	193
Période (ns)	18.6	18.5	18.5	18.6	18.5	18.5

Table 2.4 *Impact des efforts de synthèse et placement-routage pour la vitesse.*

Effort de synthèse	Normal			Fort		
Effort P&R	Standard	Moyen	Haut	Standard	Moyen	Haut
Surface (tranches)	100	100	100	100	100	100
Période (ns)	26.4	28.2	28.2	26.4	28.2	28.2

Table 2.5 *Impact des efforts de synthèse et placement-routage pour la surface.*

r–α	notation du reste partiel	repli	code de Gray	g	Surface (tranches)	Période (ns)
2-1	compl. à 2				60	23.8
4-2	compl. à 2				111	28.1
4-2	compl. à 2		✓		117	28.1
4-2	compl. à 2	✓			95	29
4-2	compl. à 2	✓	✓		101	27.4
4-3	compl. à 2				100	26.4
4-3	compl. à 2		✓		104	31.3
4-3	compl. à 2	✓			101	24.3
4-3	compl. à 2	✓	✓		104	31.6
4-3	carry-save			0	108	18.7
4-3	carry-save	✓		0	108	20.2
4-3	carry-save			1	108	20.1
4-3	carry-save			2	108	22.2
4-3	carry-save			3	108	23.1
4-3	carry-save			4	110	24
4-3	carry-save			5	111	21.6
8-4	compl. à 2				378	41.4
8-6	compl. à 2				220	33.4
8-7	compl. à 2				158	34.6
8-7	compl. à 2	✓			128	31.1
8-7	compl. à 2	✓	✓		129	31.1

Table 2.6 *Quelques exemples de l'impact des options d'optimisation pour la division SRT.*

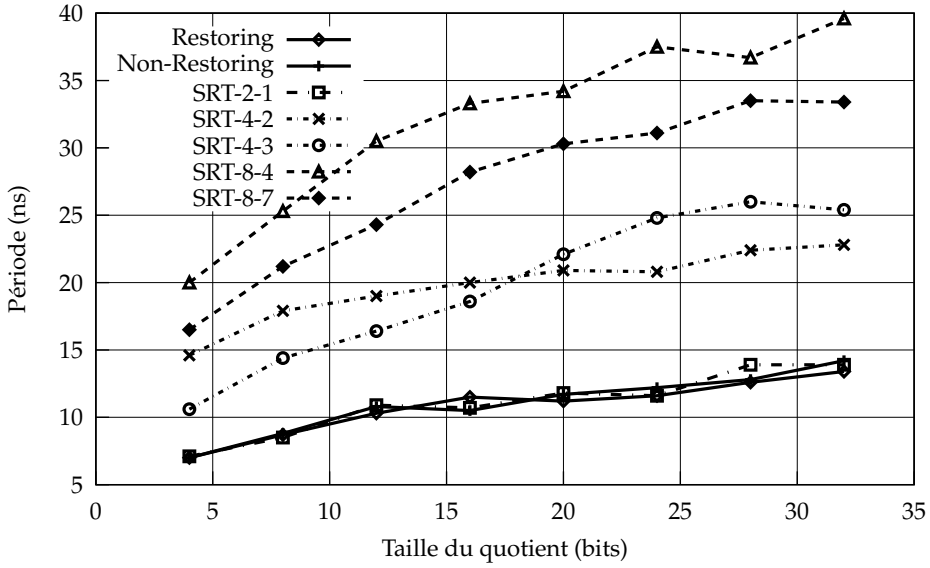

Figure 2.15 *Période d'horloge en fonction de la taille du quotient.*

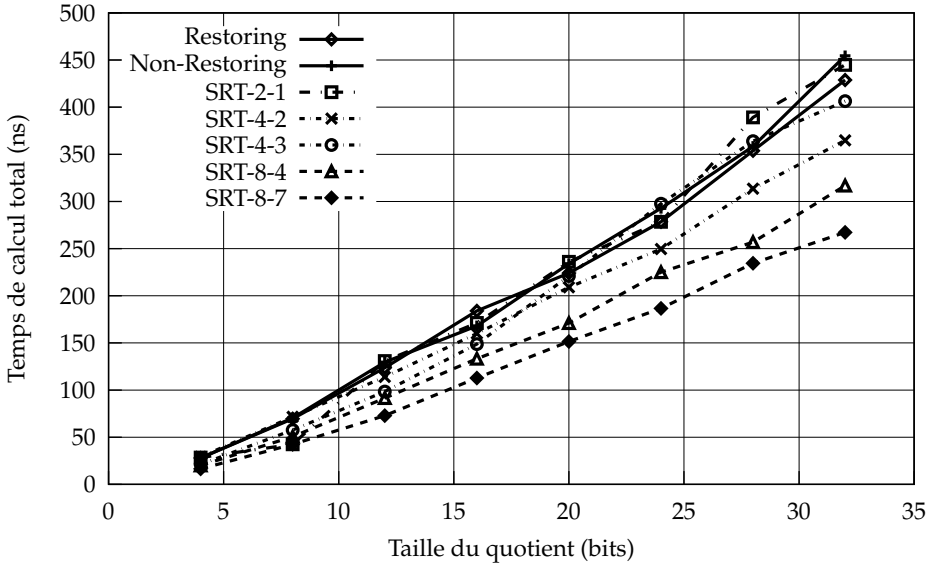

Figure 2.16 *Temps de calcul total en fonction de la taille du quotient.*

Nous avons validé le comportement des diviseurs générés à l'aide de simulations VHDL. De nombreux vecteurs aléatoires ont été testés avec succès pour tous les algorithmes, paramètres, représentations et tailles.

2.4.6 Bilan

Nous avons présenté un programme, `Divgen`, qui permet de générer automatiquement des descriptions VHDL de diviseurs optimisés. Plusieurs algorithmes et options sont disponibles. La version 0.1 de `Divgen`, qui implante les fonctionnalités décrites ici, est disponible en GPL sur le web à l'url `http ://lipforge.ens-lyon.fr/www/divgen/`.

Les résultats obtenus pour le repli de la table de sélection sont encourageants. L'utilité du code

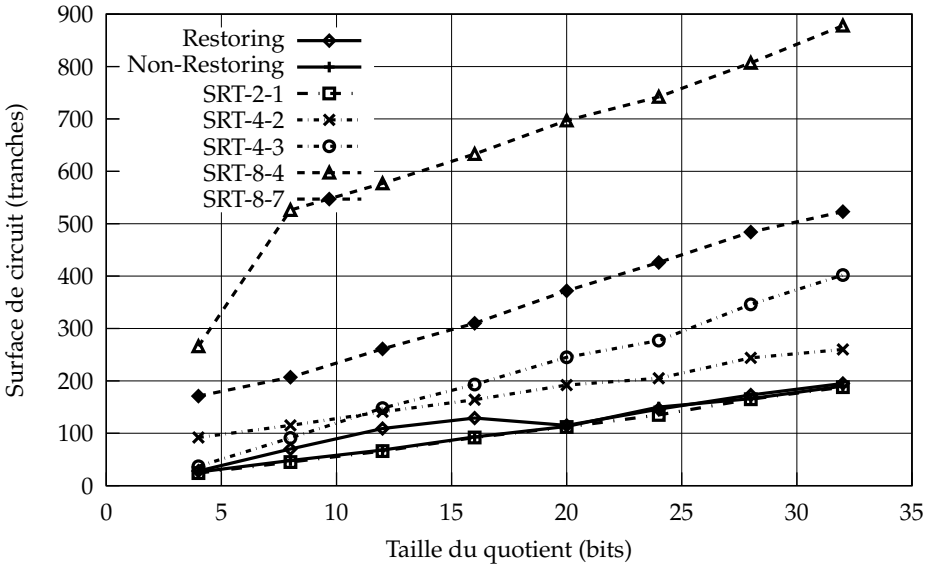

Figure 2.17 *Surface de l'opérateur en fonction de la taille du quotient.*

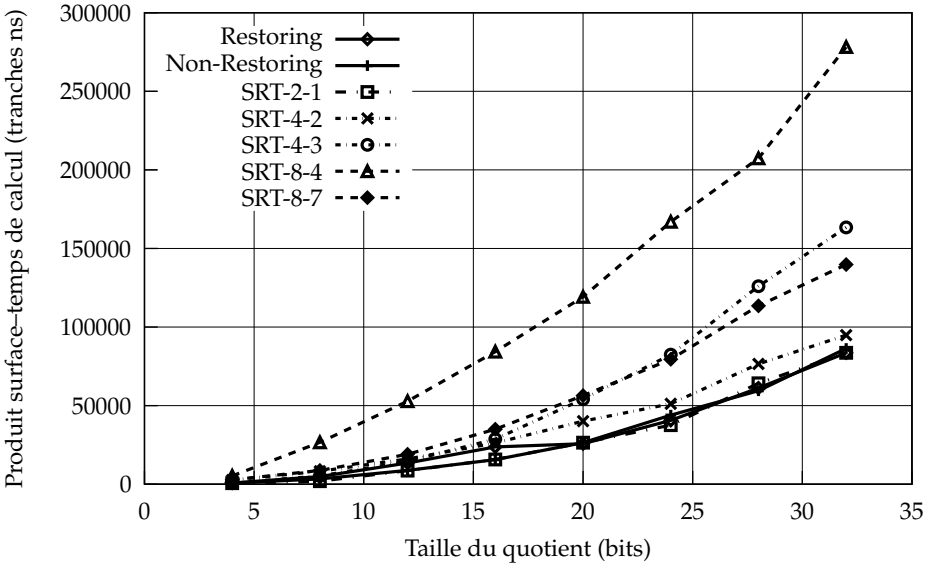

Figure 2.18 *Produit surface-temps de calcul en fonction de la taille du quotient.*

de Gray n'est pas immédiatement apparente, surtout dans le cas de tables relativement petites. Une amélioration intéressante serait de pouvoir spécifier dans des tables VHDL plusieurs valeurs acceptables pour une entrée sans devoir passer par un code de Gray.

2.5 Calcul de la racine carrée par l'algorithme SRT

L'algorithme SRT pour la division peut être étendu au calcul de la racine carrée en suivant un raisonnement proche. Cependant, sa mise en place comporte quelques « pièges » qui peuvent mener à des opérateurs incorrects, mais aussi à des fonctions de sélection plus coûteuses que nécessaire. Dans cette section, nous présentons les aspects clés d'un algorithme SRT pour l'évaluation de la racine carrée. Cette étude nous permet de mettre en évidence plusieurs aspects du développe-

ment d'algorithmes SRT qui seront utilisés lors de la conception d'un générateur automatique pour des opérateurs basés sur ces algorithmes (section 2.6).

Comme pour la division, un chiffre du résultat est produit à chaque itération par application de la récurrence, en partant des poids forts. On note $s[j]$ le résultat partiel à l'itération j.

$$s[j] = s[0] + \sum_{i=1}^{j} s_i r^{-i}.$$

Le résultat final doit vérifier :

$$0 \leq \varepsilon = \sqrt{x} - s < r^{-n}.$$

2.5.1 Extraction de l'équation de récurrence et des bornes

Le résultat est déduit du résultat partiel à l'itération n. On s'autorise une petite erreur dans le choix des chiffres du résultat partiel qui sera corrigée aux itérations suivantes. On note $\varepsilon[j]$ l'erreur à l'itération j. La convergence correcte de l'algorithme est assurée par la condition $|\varepsilon[j]| < \rho r^{-j}$, c'est à dire :

$$-\rho r^{-j} < \sqrt{x} - s[j] < +\rho r^{-j}.$$

Dans un premier temps, on essaie d'éliminer les termes « difficiles » à calculer pour extraire un reste partiel simple. Par difficile à calculer, on entend tout ce qui contient des divisions et des radicaux (des puissances rationnelles). Les termes qu'il est important de simplifier sont ceux qui ne font pas partie des bornes, c'est à dire ceux qui ne contiennent pas r^{-j}. On part de :

$$s[j] - \rho r^{-j} < \sqrt{x} < s[j] + \rho r^{-j}.$$

En élevant l'encadrement au carré, deux cas se présentent :

$$\begin{cases} \left(s[j] - \rho r^{-j}\right)^2 < x < \left(s[j] + \rho r^{-j}\right)^2 & \text{si } s[j] \geq \rho r^{-j}, \\ 0 < x < \left(s[j] + \rho r^{-j}\right)^2 & \text{si } s[j] < \rho r^{-j}. \end{cases}$$

La seconde inégalité n'est pas satisfaisante : la condition $0 < x$ est inutile. L'encadrement devient :

$$\begin{cases} -2\rho r^{-j} s[j] + \rho^2 r^{-2j} < x - s^2[j] & \text{si } s[j] \geq \rho r^{-j}, \\ x - s^2[j] < +2\rho r^{-j} s[j] + \rho^2 r^{-2j} & \text{toujours.} \end{cases}$$

On en déduit un encadrement du reste partiel $w[j] = r^j(x - s^2[j])$:

$$\underline{B}[j] < w[j] < \overline{B}[j], \quad \text{avec } \overline{B}[j] = +2\rho s[j] + \rho^2 r^{-j} \quad \text{et } \underline{B}[j] = \begin{cases} -2\rho s[j] + \rho^2 r^{-j} & \text{si } s[j] \geq \rho r^{-j}, \\ -\infty & \text{sinon.} \end{cases} \tag{2.4}$$

On note que les bornes du reste partiel pour la racine carré dépendent de l'itération, alors qu'elles ne dépendaient que de ρ et d pour la division. En utilisant les définitions du reste et du résultat partiel, on déduit l'équation de récurrence :

$$w[j] = rw[j-1] - F[j], \quad \text{où } F[j] = 2q_j s[j-1] + q_j^2 r^{-j}.$$

avec la valeur initiale $w[0] = x - s^2[0]$.

Le choix de chaque chiffre du résultat q_j doit être fait de telle façon que la borne sur le reste partiel reste vérifiée. Comme pour la division, on veut que la fonction de sélection du chiffre n'utilise que des estimations. Cette fois, cette fonction dépend du résultat partiel $s[j]$ et du reste partiel $w[j]$. D'où :

$$q_j = \text{Sel}\left(\hat{w}[j-1], \hat{s}[j-1]\right).$$

2.5.2 Fonction de sélection du chiffre q_j

On cherche maintenant une fonction de sélection qui assure un calcul valide, et qui permette un calcul rapide du nouveau chiffre q_j.

On définit pour chaque chiffre $k \in \{-\alpha, \ldots, \alpha\}$ de la représentation utilisée l'intervalle $]U_k(s)[j], L_k(s)[j][$ des valeurs de $rw[j-1]$ pour lesquelles le choix du chiffre $q_j = k$ est correct. Par définition :

$$L_k(s)[j] < rw[j-1] < U_k(s)[j] \Rightarrow \underline{B}[j] < w[j] = rw[j-1] - F[j] < \overline{B}[j].$$

On en déduit :

$$\begin{aligned} L_k(s)[j] &= \underline{B}[j] + 2ks[j-1] + k^2 r^{-j}, \\ U_k(s)[j] &= \overline{B}[j] + 2ks[j-1] + k^2 r^{-j}. \end{aligned}$$

D'où :

$$\begin{aligned} L_k(s)[j] &= \begin{cases} -2\rho s[j-1] + \rho^2 r^{-j} + 2ks[j-1] + k^2 r^{-j} & \text{si } s[j] \geq \rho r^{-j}, \\ -\infty & \text{si } s[j] < \rho r^{-j}. \end{cases} \\ U_k[j] &= 2\rho s[j] + \rho^2 r^{-j} + 2ks[j-1] + k^2 r^{-j}. \end{aligned}$$

Qui donne :

$$\begin{aligned} L_k(s)[j] &= \begin{cases} 2(k-\rho)s[j-1] + (k-\rho)^2 r^{-j} & \text{si } s[j-1] \geq (\rho-k) r^{-j}, \\ -\infty & \text{si } s[j-1] < (\rho-k) r^{-j}. \end{cases} \\ U_k[j] &= 2(k+\rho)s[j-1] + (k+\rho)^2 r^{-j}. \end{aligned}$$

Comme pour la division, la fonction de sélection est plus simple si on restreint la valeur de l'opérande à une partie du domaine d'entrée possible. Pour une représentation virgule fixe (non signée) sur le domaine $[0, 1[$, on pose :

$$s = \sqrt{x}, \qquad \frac{1}{4} \leq x < 1, \quad \text{ce qui permet : } \frac{1}{2} \leq s < 1.$$

La fonction de sélection est encore caractérisée par un ensemble de constantes. Cette fois, c'est l'estimation du résultat partiel qui est utilisée, et pas celle du diviseur comme pour la division. Les constantes $m_{k,\hat{s}}$ doivent vérifier :

$$\max_{\hat{s}=v}\left(L_k(s)[j]\right) \leq m_{k,v} \leq \min_{\hat{s}=v}\left(U_{k-1}(s)[j]\right) + \text{LSB}(w).$$

En utilisant les estimations $\hat{s}[j-1]$ et $\hat{w}[j-1]$ au lieu des valeurs exactes du résultat partiel et du reste partiel, on commet une erreur qui dépend de leur format de représentation et, pour $s[j]$, de l'itération. En effet, à l'itération j, on a $\text{LSB}(s[j]) = r^{-j}$.

La formule (2.3) de validité de la fonction de sélection s'adapte facilement :

$$\left\lceil \max_{\hat{s}=v} \left(L_k(s)[j]\right) - \underline{\delta(\hat{w})} \right\rceil_{\mathrm{LSB}(\hat{w})} \leq m_{k,v} \leq \left\lfloor \min_{\hat{s}=v} \left(U_{k-1}(s)[j-1]\right) - \overline{\delta(\hat{w})} \right\rfloor_{\mathrm{LSB}(\hat{w})} + \mathrm{LSB}(\hat{w}),$$

où $\lfloor x \rfloor_{\mathrm{LSB}(\hat{w})}$ (respectivement $\lceil x \rceil_{\mathrm{LSB}(\hat{w})}$) est x arrondi au multiple de $\mathrm{LSB}(\hat{w})$ inférieur (respectivement supérieur). $\underline{\delta(\hat{w})}$ (respectivement $\overline{\delta(\hat{w})}$) est la borne inférieure (respectivement supérieure) de l'intervalle $\delta(\hat{w})$ défini précédemment.

Fonction de sélection unique par intersections

Une difficulté supplémentaire est apparue par rapport au cas de la division : les bornes du reste partiel, donc les bornes L_k et U_k de sélection du chiffre k, dépendent de l'itération j. Pour des raisons de coût matériel, on ne veut avoir à implanter qu'une seule fonction de sélection qui devra marcher pour toutes les itérations. On va donc prendre pour chaque chiffre k, l'intersection $\bigcap_{0 \leq j \leq n}]L_k(s)[j], U_k(s)[j][$ des intervalles de validité du chiffre k pour toutes les itérations.

Les bornes L_k et U_k ne sont pas toujours fines, en particulier pour $s[j-1] < (\rho - k)r^{-j}$. On introduit un autre encadrement utile du reste partiel, basé sur sa définition $w[j] = r^j(x - s^2[j])$ et le domaine d'entrée $x \in [\frac{1}{4}, 1[$:

$$r^j(\frac{1}{4} - s^2[j]) \leq w[j] \leq r^j(1 - s^2[j]),$$

dont on déduit :

$$r^j(\frac{1}{4} - s^2[j-1]) \leq rw[j-1] \leq r^j(1 - s^2[j-1]). \tag{2.5}$$

Cet encadrement devient vite très large, mais il reste utile car c'est un encadrement des valeurs *possibles* du reste partiel, qu'il est utile de recouper avec les bornes L_k et U_k, qui quant à elles indiquent l'intervalle des valeurs pour lesquelles le *choix* du chiffre q_j est *valide*.

Par exemple en représentation 4–3, pour $j = 1$ et $s[j] \in [\frac{15}{16}, 1[$, avec $\mathrm{Est}(s) = 3$, l'encadrement (2.4) nous donne un intervalle de sélection $]L_1[1], U_1[1][$. Normalement, on intersecte cet intervalles avec ceux des autres itérations j, afin d'obtenir un intervalle pour lequel le choix $q_j = 1$ est valide tout le temps (si $s[j] \in [\frac{15}{16}, 1[$). Mais on sait d'après l'encadrement (2.5) que le reste partiel n'atteindra jamais la borne $U_1[1]$. Il est donc inutile de la prendre en compte dans l'intersection des intervalles de sélection (en pratique, on remplace les bornes L_k non atteintes par $-\infty$ et les U_k par $+\infty$).

Prolongement de la fonction de sélection

Dans [51, p. 156], les auteurs étudient un opérateur SRT calculant une racine carré qui fournit un résultat en représentation 4–3. Le reste partiel est en notation à retenue conservée. Une fonction de sélection valide est obtenue pour $j \geq 3$ par intersection des différents intervalles $]L_k(s)[j], U_k(s)[j][$. Il est encore possible d'y intégrer les intervalles de l'itération $j = 2$. Par contre, il n'est plus possible d'obtenir une fonction de sélection qui soit valide pour tous les couples d'estimations $(r\hat{w}[j-1], \hat{s}[j-1])$ si l'on ajoute les itérations $j = 1$ et $j = 0$. Les auteurs réussissent pourtant à mettre en place une fonction de sélection valide dès l'itération 0.

La raison est la suivante : lorsque le numéro d'itération j est faible, la condition de convergence $|\varepsilon[j]| < \rho r^{-j}$ autorise une latitude importante sur le résultat partiel. Par exemple, en partant de

$j = 0$ on n'arrive pas à obtenir une fonction de sélection valide pour toutes les valeurs de $\hat{s}[0]$. Cependant, on le peut pour $\hat{s}[0] = 1$. La condition de convergence impose $-\frac{2}{3} < w[0] = \sqrt{x} - s^2[0] < \frac{2}{3}$. En initialisant le résultat partiel à $s[0] = 1$, le reste partiel $w[0]$ reste donc borné quelle que soit la valeur de x. La sélection marche pour l'itération $j = 0$. À l'itération $j = 1$, les seules valeurs possibles de $s[1]$ seront $\frac{1}{2}$, $\frac{3}{4}$ et 1. Pour ces trois valeurs, la fonction de sélection est valide à partir de l'itération $j = 1$. L'encadrement du reste partiel est assez large pour permettre à ces trois « boites » de couvrir tous les cas possibles. La fonction de sélection est valide pour l'itération $j = 1$. En jouant sur le recouvrement qu'autorise la condition de convergence, les auteurs sont capables d'obtenir une fonction de sélection qui marche pour toutes les itérations, éliminant ainsi la nécessité d'une table d'initialisation.

Cette extension de la fonction de sélection a été obtenue par tâtonnements dans le cas particulier de la racine carrée. Une partie du travail réalisé dans cette thèse a été la formalisation et la généralisation de la méthode de prolongement de la fonction de sélection.

2.6 Génération d'opérateurs SRT pour certaines fonctions algébriques

En étudiant des algorithmes SRT pour plusieurs fonctions, on s'aperçoit qu'une grande partie du travail suit le même principe d'un algorithme à l'autre :

1. extraction du reste partiel et de ses bornes.
2. calcul des plages de sélection.
3. création d'une fonction de sélection valide à partir de l'itération j_0 (aussi petite que possible).
4. création d'une fonction d'initialisation, qui fournit $s[j_0]$ et calcule $w[j_0]$.
5. implantation de tables de sélection et d'initialisation, de l'équation de récurrence et de l'unité de conversion au vol. Il s'agit d'une « traduction » physique des définitions des variables et fonctions.

2.6.1 Notations spécifiques à cette section

Les opérateurs SRT dont nous étudions la génération dans cette section sont dédiés à l'évaluation d'une certaine classe de fonctions algébriques. De manière générale, une fonction f est dite algébrique sur les réels s'il existe une polynôme P non nul à coefficients réels, tel que :

$$P(X, f(X)) = 0,$$

ou $X = \{x_1, x_2, \ldots, x_m\}$ représente l'ensemble des variables de la fonction. Les algorithmes SRT permettent l'évaluation de certaines fonctions algébriques caractérisées par les polynômes P où $f(X)$ n'apparaît qu'une fois. Ceci recouvre par exemple :

- *l'inversion :* $P(x_1, f(x_1)) = 1 - x_1 f(x_1) = 0$
- *la division :* $P(x_1, x_2, f(x_1, x_2)) = x_1 - x_2 f(x_1, x_2) = 0$
- *la racine carrée :* $P(x_1, f(x_1)) = x_1 - f^2(x_1) = 0$
- *la racine carrée inverse :* $P(x_1, f(x_1)) = 1 - x_1 f^2(x_1) = 0$
- *la norme euclidienne :* $P(x_1, x_2, f(x_1, x_2)) = x_1^2 + x_2^2 - f^2(x_1, x_2) = 0$

2.6.2 Équation de récurrence

La toute première étape de la génération d'un algorithme de type SRT consiste à déduire de la condition de convergence une équation de récurrence bornée. L'équation de récurrence décrit le traitement à appliquer au reste partiel, et un encadrement de celui-ci assure la convergence du résultat partiel vers la valeur $f(X)$. Cette première étape a été implantée en Maple.

On définit le résultat partiel par :

$$s[j] = s[0] + \sum_{i=1}^{j} s_i r^{-i}.$$

Le résultat s est correct si :

$$0 \leq \varepsilon = f(X) - s < r^{-n}.$$

La récurrence doit faire converger le résultat vers cette erreur. Si on note $\varepsilon[j]$ l'erreur à l'itération j, alors :

$$\varepsilon[j] = f(X) - s[j].$$

Comme précédemment, le calcul converge vers le bon résultat si $|\varepsilon[j]| < \rho r^{-j}$. De l'erreur partielle $\varepsilon[n] \in]-\rho r^{-n}, +\rho r^{-n}[$, on se ramène à l'erreur $\varepsilon \in [0, r^{-n}[$ en examinant le signe du reste partiel : si il est négatif, alors $s = s[n] - r^{-n}$, sinon $s = s[n]$.

La condition de convergence est donc :

$$-\rho r^{-j} < f(X) - s[j] < +\rho r^{-j}.$$

Il s'agit maintenant d'extraire le reste partiel, et un encadrement sur celui-ci *équivalent* à la condition de convergence. On part donc de la condition de convergence, que l'on transforme afin d'obtenir par équivalences successives un encadrement dont le terme central est « facile » à calculer, et dont les bornes diminuent d'un facteur r à chaque itération. Plus formellement, on entend par facile à calculer que le terme central ne contient plus de divisions ni de radicaux, c'est à dire des expressions élevées à une puissance négative ou rationnelle non entière. Les bornes sont en $O(r^{-j})$, et le terme central ne contient plus de termes en r^{-j}.

Ce problème revient à déduire de l'écriture de $f(X)$ sous la forme d'un polynôme radical en X son polynôme caractéristique $P(X, f(X))$. Le fait que $f(X)$ puisse s'exprimer sous la forme d'un polynôme radical en X est la raison pour laquelle on s'est restreint aux fonctions algébriques caractérisées par un polynôme où $f(X)$ n'apparaît qu'une seule fois.

On transforme l'inéquation de convergence en :

$$s[t] - \rho r^{-j} < f(X) < s[j] + \rho r^{-j}. \tag{2.6}$$

Élimination des divisions

Dans un premier temps, on élimine les divisions qui apparaissent dans l'inéquation. Il est toujours possible d'écrire $f(X)$ comme une fraction irréductible de deux fonctions

$$f(X) = \frac{a(X)}{b(X)},$$

où ni $a(X)$ ni $b(X)$ ne contiennent de divisions. On étudie $b(X)$ sous une représentation de la forme

$$b(X) = (c(X) + d(X))^{1/e},$$

où $c(X)$ est un polynôme multivarié en X sans radicaux, $d(X)$ est le radical d'un polynôme radical en X, et e un entier non nul aussi petit que possible (il peut valoir 1).

On peut transformer l'inéquation (2.6) en :

$$b^e(X)(s[j] - \rho r^{-j})^e < a^e(X) < b^e(X)(s[j] + \rho r^{-j})^e.$$

Cette condition est valide si e est impair, ou si $s[j] \geq \rho r^{-j}$. L'inéquation devient :

$$b^e(X)(s[j] - \rho r^{-j})^e - d(X)s^e[j] < a^e(X) - d(X)s^e[j] < b^e(X)(s[j] + \rho r^{-j})^e - d(X)s^e[j].$$

Notée par la suite :

$$\underline{B_1} < f_1(X) < \overline{B_1}. \tag{2.7}$$

Les bornes $\underline{B_1}$ et $\overline{B_1}$ se développent toutes deux sous la forme $c(X)s^e[j] + O(r^{-j})$. Si la condition $s[j] \geq \rho r^{-j}$ n'est pas vérifiée, alors la borne $\underline{B_1} < f_1(X)$ va être inutile, comme on l'a vu dans le cas de la racine carrée. On pose alors $\underline{B_1} = -\infty$.

Dans le cas où $f(X)$ ne contient pas de divisions, on prend $c(X) = 1$, $d(X) = 0$ et $e = 1$: les inéquations (2.6) et (2.7) sont alors identiques.

Suite à cette transformation, on peut assurer que quelle que soit la puissance à laquelle on élève (2.7), les bornes de l'équation résultante ne contiendront que des termes sans divisions, ce qui sera utile lors de l'extraction finale du reste partiel.

Notre programme demande à la fonction passée en paramètre d'être déjà écrite sous la forme $a(X)/(c(X) + d(X))^{(1/e)}$.

Exemple 2.6 *Si on cherche à calculer un opérateur pour la fonction*

$$f(X) = \frac{1}{\sqrt{1 + \sqrt{\frac{x_2}{x_1}}}},$$

on commence par transformer l'équation de convergence en :

$$f(X) = \frac{\sqrt{\sqrt{x_1}}}{\sqrt{\sqrt{x_1} + \sqrt{x_2}}} = \frac{\sqrt{x_1}}{\sqrt{x_1 + \sqrt{x_1 x_2}}}.$$

On a donc $a(X) = \sqrt{x_1}$, $b(X) = \sqrt{x_1 + \sqrt{x_1 x_2}}$, $c(X) = x_1$, $d(X) = \sqrt{x_1 x_2}$ *et* $e = 2$.
L'encadrement calculé par le programme est alors

$$(x_1 + \sqrt{x_1 x_2})(s[j] - \rho r^{-j})^2 < x_1 < (x_1 + \sqrt{x_1 x_2})(s[j] + \rho r^{-j})^2,$$

qui devient

$$\begin{aligned} x_1 s^2[j] + (x_1 + \sqrt{x_1 x_2})(-2\rho s[j] r^{-j} + \rho^2 r^{-2j}) \\ < x_1 - \sqrt{x_1 x_2} s^2[j] \\ < x_1 s^2[j] + (x_1 + \sqrt{x_1 x_2})(+2\rho s[j] r^{-j} + \rho^2 r^{-2j}). \end{aligned}$$

L'inéquation ne contient plus de divisions, et les termes contenant des radicaux qui ne sont pas en $O(r^{-j})$ *on tous été ramenés au centre.*

Élimination des radicaux

Il reste maintenant à manipuler l'encadrement (2.7) pour ne laisser que des termes « faciles » à calculer au centre, et des bornes en $O(r^{-j})$.

La méthode proposée repose sur le fait qu'il n'existe qu'un nombre fini de radicaux pouvant apparaître dans un polynôme à coefficients entiers en $\{X, f_1(X)\}$. Ces radicaux peuvent être calculés à partir de $f_1(X)$.

Considérons par exemple la fonction $f(X) = \sqrt{x_1} + \sqrt[3]{x_2}$. On veut éliminer les radicaux de l'encadrement sans divisions obtenu (trivialement) à l'issue de l'étape précédente :

$$s[j] - \rho r^{-j} < \sqrt{x_1} + \sqrt[3]{x_2} < s[j] + \rho r^{-j}.$$

On veut par additions, multiplications et élévations à des puissances arriver à un encadrement sans radicaux dans les termes qui ne sont pas en $O(r^{-j})$. Les radicaux qui apparaissent dans f_1 sont $\{\sqrt{x_1}, \sqrt[3]{x_2}\}$. Lorsque l'on élève $f_1(X)$ à une puissance entière, de nouveaux radicaux apparaissent, en nombre fini.

L'idée est d'étudier le problème du point de vue des espaces vectoriels. Posons $\mathbf{e_0} = 1$, $\mathbf{e_1} = \sqrt{x_1}$ et $\mathbf{e_2} = \sqrt[3]{x_2}$ des vecteurs d'un espace vectoriel à coefficients polynomiaux en X. Par produits vectoriels successifs, on arrive à obtenir une base complète $(\mathbf{e_0}, \mathbf{e_1}, \ldots, \mathbf{e_5})$. La table des produits (commutatifs) de vecteurs est la suivante :

	$\mathbf{e_0}$	$\mathbf{e_1}$	$\mathbf{e_2}$	$\mathbf{e_3}$	$\mathbf{e_4}$	$\mathbf{e_5}$
$\mathbf{e_0} = 1$	$\mathbf{e_0}$	$\mathbf{e_1}$	$\mathbf{e_2}$	$\mathbf{e_3}$	$\mathbf{e_4}$	$\mathbf{e_5}$
$\mathbf{e_1} = \sqrt{x_1}$		$x_1\mathbf{e_0}$	$\mathbf{e_4}$	$\mathbf{e_5}$	$x_1\mathbf{e_2}$	$x_1\mathbf{e_3}$
$\mathbf{e_2} = \sqrt[3]{x_2}$			$\mathbf{e_3}$	$\mathbf{e_5}$	$x_2\mathbf{e_1}$	
$\mathbf{e_3} = \sqrt[3]{x_2}^2$				$x_2\mathbf{e_2}$	$x_2\mathbf{e_1}$	$x_2\mathbf{e_4}$
$\mathbf{e_4} = \sqrt{x_1}\sqrt[3]{x_2}$					$x_1x_2\mathbf{e_2}$	$x_1x_2\mathbf{e_0}$
$\mathbf{e_5} = \sqrt{x_1}\sqrt[3]{x_2}^2$						$x_1x_2\mathbf{e_2}$

Il est maintenant possible d'exprimer un polynôme en $\{X, f_1(X)\}$ sous la forme d'un vecteur de la base que l'on vient de constituer.

Exemple 2.7 *Pour* $f_1(X) = \sqrt{x_1} + \sqrt[3]{x_2}^2$ *:*

$$\begin{aligned}(3x_1+1)f_1(X) - f_1^3(X) &= 2\sqrt{x_1}^3 + \sqrt{x_1} + \sqrt[3]{x_2} - 3\sqrt{x_1}\sqrt[3]{x_2}^2 - x_2 \\ &= -x_2 + (2x_1+1)\sqrt{x_1} + \sqrt[3]{x_2} - 3(\sqrt{x_1}\sqrt[3]{x_2}^2) \\ &\Leftrightarrow -x_2\mathbf{e_0} + (2x_1+1)\mathbf{e_1} + \mathbf{e_2} - 3\mathbf{e_5}.\end{aligned}$$

L'expression précédente se lit comme un vecteur de coordonnées $(-x_2, 2x_1+1, 1, 0, 0, -3)$.

En traitant chaque radical comme une dimension d'un espace vectoriel à coefficients polynomiaux en X, notre problème se ramène à constituer une base de cet espace vectoriel, puis à calculer un vecteur non nul colinéaire au vecteur $\mathbf{e_0}$. On constitue cette base en calculant les différentes puissances entières de $f_1(X)$. Il suffit ensuite d'effectuer une triangularisation de Gauss partielle

(avec des coefficients polynomiaux en X) sur la matrice constituée. Pour simplifier, on effectue des pseudo-divisions.

Afin de calculer une solution pour la fonction $f_1(X) = \sqrt{x_1} + \sqrt[3]{x_2}^2$, on constitue l'ensemble de vecteurs suivant :

$$\begin{bmatrix} f_1(X)^1 \\ f_1(X)^2 \\ f_1(X)^3 \\ f_1(X)^4 \\ f_1(X)^5 \\ f_1(X)^6 \end{bmatrix} \Leftrightarrow \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ x_1 & 0 & 0 & 1 & 2 & 0 \\ x_2 & x_1 & 3x_1 & 0 & 0 & 3 \\ x_1^2 & 4x_2 & x_2 & 6x_1 & 4x_1 & 0 \\ 10x_1x_2 & x_1^2 & 5x_1^2 & x_2 & 5x_2 & 10x_1 \\ x_1^3 + x_2^2 & 20x_1x_2 & 15x_1x_2 & 15x_1^2 & 6x_1^2 & 6x_2 \end{bmatrix}$$

Après trigonalisation on obtient le vecteur $(x_1^3 - x_2^2, 0, 0, 0, 0, 0)$, colinéaire à $\mathbf{e_0} = 1$ via la transformation

$$f(X) \rightarrow -6x_1x_2f(X) + 3x_1^2f^2(X) - 2x_2f^3(X) - 3x_1f^4(X) + f^6(X).$$

Si on assure que $\underline{B_1} \geq 0$ dans le cas où $f(X)$ apparaît élevé à une puissance paire, et en ayant des garanties strictes sur le signe des coefficients multipliant les $f^i(X)$ dans la transformation, il est possible de déduire un encadrement sur un terme central simple, qui assure la convergence du calcul. Ici, si on garantit que $x_1 \geq 0$ et $x_2 \geq 0$ sur les domaines d'utilisation (ce qui constitue une hypothèse raisonnable), alors on peut déduire :

$$\begin{aligned} -6x_1x_2\overline{B_1} + 3x_1^2\underline{B_1}^2 - 2x_2\overline{B_1}^3 - 3x_1\overline{B_1}^4 + \underline{B_1}^6 \\ < x_1^3 - x_2^2 \\ < -6x_1x_2\underline{B_1} + 3x_1^2\overline{B_1}^2 - 2x_2\underline{B_1}^3 - 3x_1\underline{B_1}^4 + \overline{B_1}^6. \end{aligned}$$

On note le nouvel encadrement

$$\underline{B_2} < f_2(X) < \overline{B_2}.$$

Extraction du reste partiel

On sait que le terme central est simple à calculer, et que tous les termes apparaissant dans les bornes sont soit simples, soit multiples de r^{-j}. De plus tous les termes simples (en $s[j]$) apparaissaient à l'identique dans les deux bornes $\underline{B_1}$ et $\overline{B_1}$, c'est aussi le cas dans $\underline{B_2}$ et $\overline{B_2}$. On peut donc les ramener au centre. Les bornes sont en $O(r^{-j})$, on multiplie le tout par r^j pour obtenir le reste partiel. Pour notre exemple, on obtient :

$$L[j] < w[j] = r^j(x_1^3 - x_2^2 + 6x_1x_2s[j] - 3x_1^2s[j]^2 + 2x_1s[j]^3 + 3x_1s[j]^4 - s[j]^6) < U[j].$$

L'équation de récurrence est ensuite facilement obtenue par substitution dans la définition de $w[j]$ du résultat partiel $s[j]$ par $s[j-1] + q_jr^{-j}$. On arrive à l'équation de récurrence $w[j] = rw[j-1] - F_{q_j}[j]$.

Le calcul des bornes $]L_k, U_k[$ de sélection du chiffre $q_j = k$ est immédiat :

$$\begin{aligned} L_k[j] &= L[j] + F_k[j], \\ U_k[j] &= U[j] + F_k[j]. \end{aligned}$$

Implantation logicielle

L'extraction de reste partiel est implantée à l'aide d'un ensemble de fonctions Maple, pour un total de 250 lignes environ. Nous l'avons testée pour toutes les fonctions algébriques usuelles, et pour des fonctions plus compliquées comme celles présentées en exemple.

Dans la mesure où Maple est seulement utilisé pour la manipulation d'expressions (on n'appelle pas de fonctions complexes telles que `solve` par exemple), une traduction en C++ à l'aide d'une bibliothèque libre de manipulation d'expressions symboliques telle que Yacas ou Ginac devrait être assez simple.

2.6.3 Calcul de la fonction de sélection

La fonction de sélection d'un algorithme SRT prend comme paramètres une estimation des variables de X, du résultat partiel $s[j-1]$ et du reste partiel $w[j-1]$. Elle retourne un nouveau chiffre $q_j \in \{-\alpha, \ldots, +\alpha\}$ du résultat partiel. La fonction de sélection doit assurer que si le reste partiel est borné à l'itération $j-1$, alors le chiffre q_j retourné assure que le nouveau reste partiel $w[j] = rw[j-1] - F[j]$ sera aussi borné. On a :

$$q_j = \text{Sel}\left(\hat{X}, \hat{s}[j-1], r\hat{w}[j-1]\right).$$

Nous décrivons ici l'algorithme que nous avons implanté pour le calcul de fonctions de sélection peu coûteuses en termes de matériel. Cet algorithme est écrit en Maple, et représente environ 1500 lignes de code. Il lui reste encore à subir des tests, mais il a déjà permis d'obtenir des fonctions de sélection de coût matériel faible pour la division, la racine carrée et la racine carrée inverse. La fonction de sélection est généralement implantée à l'aide d'une table, qui lit en entrée quelques-uns des premiers bits de X, $s[j]$ et $w[j]$.

Pour obtenir une fonction de sélection, on effectue un parcours sur le coût des estimations de X et du résultat partiel $s[j-1]$. Pour chaque découpage, on cherche à obtenir une fonction de sélection valide. Si une telle fonction existe, elle correspond à une précision minimale $\text{Est}(w)$ de l'estimation sur le reste partiel $w[j-1]$. Plusieurs solutions sont retournées, qui réalisent un compromis entre $\text{Est}(w)$ et l'itération j_0 à partir de laquelle la fonction de sélection est valide. En effet, le coût de la fonction de sélection est d'autant plus grand que $\text{Est}(w)$ est élevé, mais si c'est j_0 qui est grand, on le paiera au niveau de la fonction d'initialisation de la récurrence.

Le parcours sur les estimations de X et du reste partiel $s[j-1]$ est un simple comptage dans $\mathbb{N}^{m+1}$, où m est le nombre de variables de la fonction $f(X)$.

La suite décrit notre algorithme pour le calcul de fonctions de sélection. Ce calcul est fait pour un découpage $(\text{Est}(x_1), \ldots, \text{Est}(x_m), \text{Est}(s))$ donné. On décrit successivement :

1. la notion de boite, qui servira par la suite.
2. le calcul réalisé par la fonction `ComputeWRange`, qui pour une itération j donnée et pour chaque « boite » correspondant à une entrée sur $(\hat{X}, \hat{s}[j-1])$, donne pour chaque choix du chiffre $q_j = k$ possible à quel intervalle réel le reste partiel doit appartenir pour que ce choix soit correct.
3. la fonction `WRangeToWEst`, qui calcule en utilisant les intervalles calculés par `ComputeWRange` quelle estimation du reste partiel permet d'obtenir une fonction de sélection valide.
4. ce que l'on entend par « couverture », et la façon dont celles-ci sont calculées par la fonction `TableCovers`.

5. Enfin, nous présenterons l'algorithme qui utilise les fonctions précédentes pour calculer des fonctions de sélection valides, pour un découpage donné sur les variables de X et le résultat partiel.

Notion de « boite »

Dans la suite, on appellera « boite » une combinaison d'intervalles sur les variables x_1 à x_m et sur le résultat partiel $s[j-1]$. Ces intervalles correspondent à une estimation constante de ces valeurs. C'est à dire qu'une boite est le produit cartésien des intervalles sur les variables et le résultat partiel dans lequel $(\hat{X}, \hat{s}[j-1])$ reste constant.

Par exemple, pour $X = \{x_1\}$ avec $x_1 \in [\frac{1}{4}, 1[$ et $s[j-1] \in [\frac{1}{2}, 1]$, si on a un découpage avec $\text{Est}(x_1) = 1$ et $\text{Est}(s) = 3$, il existe 8 boites. La boite $(\frac{1}{2}, \frac{3}{4})$ correspond à des intervalles $(x_1, s[j-1]) \in [\frac{1}{2}, 1[\times[\frac{3}{4}, \frac{13}{16}[$.

Pour chaque boite possible, connaissant les domaines des variables et du résultat partiel, le programme commence par pré-calculer les intervalles de valeurs qui s'y rattachent. Les intervalles calculés sont fins, c'est à dire que l'intervalle pour x_1 dans l'intervalle précédent sera en fait $[\frac{1}{2}, 1 - \text{LSB}(x_1)]$. Le résultat partiel est un cas spécial. D'une part, il atteint sa borne supérieure, que l'on traite à part. D'autre part, la largeur de l'intervalle correspondant à une valeur de l'estimation du résultat partiel dépend de l'itération, car $\text{LSB}(s) = r^{-j}$. Lors du calcul de l'intervalle, on aura $s[j-1] \in [\frac{3}{4}, \frac{13}{16} - \min(\text{LSB}(s), \text{LSB}(\hat{s}))]$.

Plages de sélection à j donné

Le but de la fonction `ComputeWRange` est le calcul, pour une itération j donnée et pour chaque boite, c'est à dire pour chaque combinaison d'estimations sur les x_i et sur $s[j-1]$ possible, pour quels intervalles de $rw[j-1]$ on peut choisir un chiffre $q_j = k$, et quelles valeurs $rw[j-1]$ peut atteindre.

On parcourt donc toutes les boites. Dans un premier temps, on regarde si la boite peut être utilisée à l'itération j. Il faut évidemment qu'elle corresponde à des intervalles non vides des entrées (pensez par exemple à la boite $\hat{x_1} = 0$, où $x_1 \in [\frac{1}{4}, 1[$ et $\text{Est}(x_1) = 2$). Il faut aussi que cette boite puisse être utilisée au regard de la condition de convergence : les intervalles des x_i correspondant à cette boite définissent un intervalle possible pour $f(X)$. En notant $[\underline{s}, \overline{s}[$ l'intervalle du résultat partiel pour cette boite, alors elle n'est utilisée que si l'intervalle $[\underline{s} - \rho r^{-j}, \overline{s} + \rho r^{-j}[$ a un intersection non nulle avec l'intervalle des $f(X)$ possibles. Si on oublie de filtrer ces boites, on n'aura pas pour autant des fonctions de sélection fausses. Par contre on risque d'obtenir des fonctions plus coûteuses ou de perdre des solutions, car on ajoute des contraintes inutiles sur certaines boites.

Si la boite est utilisée, on parcourt les chiffres $k \in \{-\alpha, \ldots, +\alpha\}$. Deux cas se présentent : soit on veut obtenir des plages valides pour j seulement, soit on veut des plages qui marchent pour toutes les itérations *à partir* de j.

– Si on cherche une plage pour j seulement, on sait que le choix du chiffre k pour la boite sera correct lorsque $L_k[j] < rw[j-1] < U_k[j]$. On effectue un calcul fin par intervalles des bornes $L_k[j]$ et $U_k[j]$ pour la boite donnée. Ceci se fait en Maple en utilisant la fonction `evalr`. En notant les intervalles obtenus $[\underline{L_k}, \overline{L_k}]$ et $[\underline{U_k}, \overline{U_k}]$ respectivement, un encadrement des valeurs de $rw[j-1]$ (pas de son estimation) pour lesquelles $q_j = k$ est correct est :

$$\overline{L_k} < rw[j-1] < \underline{U_k}.$$

Pour obtenir des plages valables pour toutes les itérations *en pratique*, il suffit de prendre l'intersection des plages pour j allant de j_0 à n, où n est la dernière itération de l'algorithme, celle où l'erreur partielle $|\varepsilon[n]| < r^{-n}$ est atteinte.

– Si calculer les plages pour chaque itération indépendamment est impossible, alors il faut calculer des plages de valeurs qui restent valides pour toutes les itérations à partir de j. C'est par exemple ce qui est fait dans [51] pour le calcul de la racine carrée.
Les bornes $L_k[j]$ et $U_k[j]$ varient au cours des itérations. Une table qui fonctionne pour l'itération j peut donc ne plus être valable à l'itération suivante. Afin de contourner cet obstacle, on va introduire de nouvelles bornes dérivées $L_k^+[j]$, $L_k^-[j]$, $U_k^+[j]$ et $U_k^-[j]$. Ces bornes dépendent de la boite considérée. On sait que la variation des bornes au cours des itérations est due aux termes en $r^{-j}, r^{-2t}, r^{-3t}, \ldots$ qui apparaissent dans les bornes, et dont la valeur décroît strictement en amplitude. Nous allons factoriser les termes de la borne considérée selon les différentes puissances r^j. Par exemple $L_k[j] = l_0 + l_1 r^{-j} + l_2 r^{-2j} + l_3 r^{-3j} + \ldots$ Les l_i sont sont des fonctions de X et $s[j-1]$. Ceux qui décroissent au cours des itérations ($i > 0$) sont évalués par intervalles pour la boite considérée. Pour obtenir $L_k^-[j]$, on va remplacer dans la borne $L_k[j]$ tous les termes l_i, $i > 0$ par le minimum entre 0, et la borne inférieure de leur évaluation par intervalle. Pour $L_k[j]^+$, on les remplacera par le maximum entre 0 et la borne supérieure de l'évaluation. On effectue le même traitement sur U_k pour obtenir $U_k[j]^+$ et $U_k[j]^-$.
Ces bornes « pessimistes » peuvent exagérer l'effet des termes l_i. Par exemple si $L_k[j] = x^3 2^{-j} - x^3 2^{-2j}$, dans une boite correspondant à $x_1 \in [-1, 1]$, alors $L_k^-[0] = -2$, et $L_k^-[1] = -3/4$. La vraie borne inférieure pour $j = 0$ et $j = 1$ est $-\frac{1}{4}$ (qui correspond à $L_k[1]$ pour $x_1 = -1$), les bornes pessimistes sont donc larges. Mais si on avait pris $L_k[j]$ directement, alors on aurait une borne inférieure en $t = 0$ de 0, qui n'est plus valable pour les $t > 0$. Une table basée sur cette borne serait donc fausse.

Les bornes L_k et U_k ne sont pas toujours fines, en particulier pour $s[j-1] < (\rho - k)r^{-j}$. Comme on l'a vu pour la racine carrée, il peut être intéressant d'introduire un autre encadrement du reste partiel, basé sur sa définition et les intervalles d'entrée correspondant à la boite étudiée. Pour ce faire, on effectue une évaluation fine par intervalles de la valeur $w[j] = r^j(\cdots)$.

Cet encadrement devient vite très large, mais il est utile car il donne un encadrement des valeurs *possibles* du reste partiel, qui peut être recoupé avec les bornes L_k et U_k, qui quant à elles indiquent l'intervalle des valeurs pour lesquelles le *choix du chiffre q_j est valide*.

On rappelle l'exemple donné pour la racine carrée : en représentation 4–3, pour $j = 1$ et $s[j] \in [\frac{15}{16}, 1[$ (Est$(s) = 3$), l'encadrement (2.4) nous donne un intervalle de sélection $]L_1[1], U_1[1][$. Normalement, on intersecte cet intervalle avec ceux des autres itérations j, afin d'obtenir un intervalle pour lequel le choix $q_j = 1$ est valide tout le temps (si $s[j] \in [\frac{15}{16}, 1[$). Mais on sait d'après l'encadrement introduit que le reste partiel n'atteindra jamais la borne $U_1[1]$. Il est donc inutile de la prendre en compte dans l'intersection des intervalles de sélection (en pratique, on remplace les bornes L_k non atteintes par $-\infty$ et les U_k par $+\infty$).

Par la suite, on note $[\underline{sel_k}, \overline{sel_k}]$ l'intervalle des valeurs de $rw[j-1]$ pour lesquelles le choix $q_j = k$ est correct.

En parallèle des plages de sélection, on calcule aussi à l'aide des bornes $L_{-\alpha}$ et $U_{+\alpha}$, et de l'encadrement de valeurs possibles du reste partiel, quelles sont les valeurs extrêmes qu'il peut atteindre. L'encadrement obtenu sera utilisé pour connaître le bit de poids fort (ou MSB pour *Most Significant Bit*) de l'estimation du reste partiel.

Format de y	$\delta(\hat{y}) - \hat{y}$
Complément à 2	$[0, \text{LSB}(\hat{y}) - \text{LSB}(y)]$
Carry-Save	$[0, (1 + 2^{-g})\text{LSB}(\hat{y}) - 2\ \text{LSB}(y)]$
Borrow-Save	$[-2^{-g}\text{LSB}(\hat{y}) + \text{LSB}(y), \text{LSB}(\hat{y}) - \text{LSB}(y)]$

Table 2.7 *Erreur $\delta(\hat{y})$ faite en estimant y par $\hat{y}$ (idem table 2.1).*

Paramètres d'estimation du reste partiel

Connaissant pour une boite donnée les plages de sélection de chaque chiffre $q_j = k$ et la dynamique du reste partiel, la fonction `WRangeToWEst`, calcule l'estimation la plus grossière qui permet le calcul correct de la fonction de sélection pour cette boite. On reporte la table des erreurs d'estimations δ (table 2.7). La condition que l'on a donnée pour l'encadrement des constantes m_k pour la division et la racine carrée reste valide dans le cas général. Les constantes $m_{k,(\hat{X},\hat{s})}$ doivent vérifier :

$$\max_{\hat{X}=v,\hat{s}=w} (L_k[j]) \leq m_{k,(v,w)} \leq \min_{\hat{X}=v,\hat{s}=w} (U_{k-1}[j]) + \text{LSB}(w).$$

Les valeurs $\max_{\hat{X}=v,\hat{s}=w} (L_k[j])$ et $\min_{\hat{X}=v,\hat{s}=w} (U_{k-1}[j])$ ont été calculées par la fonction `ComputeWRange`. Elles sont respectivement notées, pour une boite donnée, $\underline{sel_k}$ et $\overline{sel_k}$.

La formule (2.3) de validité de la fonction de sélection dans le cas d'estimations reste la même que précédemment :

$$\left\lceil \underline{sel_k} - \underline{\delta(\hat{w})} \right\rceil_{\text{LSB}(\hat{w})} \leq m_{k,(\hat{X},\hat{s})} \leq \left\lfloor \overline{sel_k} - \overline{\delta(\hat{w})} \right\rfloor_{\text{LSB}(\hat{w})} + \text{LSB}(\hat{w}), \tag{2.8}$$

où $\lfloor x \rfloor_{\text{LSB}(\hat{w})}$ (respectivement $\lceil x \rceil_{\text{LSB}(\hat{w})}$) est x arrondi au multiple de $\text{LSB}(\hat{w})$ inférieur (respectivement supérieur). $\underline{\delta(\hat{w})}$ (respectivement $\overline{\delta(\hat{w})}$) est la borne inférieure (respectivement supérieure) de l'intervalle $\delta(\hat{w})$.

Pour déterminer $\text{LSB}(\hat{w})$, l'algorithme part d'une valeur élevée. Si une notation redondante du reste partiel est utilisée, le nombre de bits de garde g est calculé pour que l'on ait toujours $2^{-g}\text{LSB}(\hat{w}) = \text{LSB}(w)$, c'est à dire que l'on se ramène au cas du complément à 2. Ensuite, on divise $\text{LSB}(\hat{w})$ par 2, jusqu'à ce qu'il soit possible de trouver une constante $m_{k,(\hat{X},\hat{s})}$ qui convienne. C'est à dire que l'on le divise par 2 tant que :

$$\left\lceil \underline{sel_k} - \underline{\delta(\hat{w})} \right\rceil_{\text{LSB}(\hat{w})} > \left\lfloor \overline{sel_k} - \overline{\delta(\hat{w})} \right\rfloor_{\text{LSB}(\hat{w})} + \text{LSB}(\hat{w}).$$

Si on ne sort pas de boucle, même pour $\text{LSB}(\hat{w}) = \text{LSB}(w)$, on en conclut que la fonction de sélection dans cette boite ne peut pas fonctionner.

Une fois sorti de cette boucle, si une notation redondante est utilisée, l'algorithme pose $g = 0$ puis l'incrémente jusqu'à ce que la condition ci-dessus soit de nouveau vérifiée.

Lorsque l'on veut replier la table, comme peut le faire le programme `Divgen`, la condition devient :

$$\left\lceil \max(\underline{sel_k}, \overline{sel_{-k}}) - \underline{\delta(\hat{w})} \right\rceil_{\text{LSB}(\hat{w})} > \left\lfloor \min(\overline{sel_k}, \underline{sel_{-k}}) - \overline{\delta(\hat{w})} \right\rfloor_{\text{LSB}(\hat{w})} + \text{LSB}(\hat{w}).$$

Ensuite on peut calculer le bit de poids le plus fort de l'estimation $r\hat{w}[j-1]$, en se servant de l'intervalle extrême des valeurs du reste partiel calculé par la fonction `ComputeWRange`.

La même estimation doit être utilisée pour toutes les boites de la fonction de sélection. Pour deux boites a et b, si on a besoin d'une estimation du reste partiel qui va des bits de poids a_{MSB} à a_{LSB} (avec éventuellement a_g bits de garde) pour a d'une part, de b_{MSB} à b_{LSB} (avec b_g bits de garde) pour b d'autre part part, alors une estimation qui marche pour les deux boites lit les bits de poids $\max(a_{MSB}, b_{MSB})$ à $\min(a_{LSB}, b_{LSB})$, et utilise g bits de garde, de telle façon que $2^{-g}\min(a_{LSB}, b_{LSB}) = \min(2^{-g_a}a_{LSB}, 2^{-g_b}b_{LSB})$.

Couverture de la fonction de sélection

Comme on l'a vu dans le cas de la racine carrée, il n'est pas toujours nécessaire que toutes les boites de la fonction de sélection soient valides. Considérons trois boites correspondant à des estimations $\hat{s}[j-1]$ du résultat partiel successives. Supposons que celle du milieu n'est pas valide, mais que les deux autres le sont. Si la marge autorisée par la condition de convergence est suffisante, la boite du milieu est « couverte », et la fonction de sélection peut être globalement valide dans la mesure où l'initialisation (ou les étapes précédentes) assurent que la boite du milieu ne sera jamais utilisée.

Le but de la fonction `TableCovers` est de calculer de telles couvertures. Elle retourne un ensemble de couvertures possibles, chacune associée à un coût d'estimation du reste partiel. En effet, plus on fait une estimation précise du reste partiel, plus on a de chances qu'une boite soit valide. À coût d'estimation égal, la couverture retournée est celle qui contient le plus de boites valides.

Fonctions de sélection possibles

L'algorithme qui calcule, pour des estimations données sur les variables de X et le résultat partiel $s[j-1]$, des fonctions de sélection efficaces fonctionne de la façon suivante. Dans un premier temps, on calcule les plages de sélection à l'aide de la fonction `ComputeWRange`. On prend leurs intersections, en remontant depuis la dernière itération n vers l'itération 0. De ces plages de valeurs, on déduit pour chaque boite une estimation requise sur le reste partiel. Lorsque la condition de convergence le permet, on calcule les couvertures possibles de la fonction de sélection.

On va alors créer à partir de ces éléments une liste contenant une itération de départ, un schéma d'estimation du reste partiel, une couverture indiquant quelles boites (potentiellement toutes) sont valides lors de la première itération, et pour chaque boite les plages de valeurs du reste partiel pour lesquelles le choix du chiffre $q_j = k$ est valide. Il ne reste plus qu'à traduire ces plages en entrées, toujours à l'aide de l'encadrement (2.8).

L'algorithme pour le calcul de fonctions de sélection est écrit en Maple. Il a été testé dans quelques cas pour des algorithmes calculant la division, la racine carrée et la racine carré inverse. Les options comme le repli de la table et la saturation du reste partiel sont implantées. Nous sommes actuellement en train d'ajouter la gestion d'un cas particulier : lorsque l'on travaille avec un reste partiel en notation redondante, en particulier le carry-save, il se peut que le reste partiel ne soit pas vraiment en représentation redondante lors de l'itération j_0 (il est représenté à l'aide des bits s_i du carry save, les bits c_i restent à 0). Tirer parti de ce cas peut permettre de gagner une itération sur la fonction de sélection.

2.6.4 État actuel des travaux

À ce jour, le programme implantant nos algorithmes pour la génération d'opérateurs SRT est capable d'extraire l'équation de récurrence, et les expressions des plages $]L_k, U_k[$. Le calcul de la fonction de sélection est implanté et en cours de test. Nous avons plusieurs algorithmes pour une fonction d'initialisation à base de tables, qui restent à écrire. Il pourrait aussi être intéressant d'étudier la précision requise pour une initialisation à l'aide de polynômes d'approximation. L'étape de traduction vers le matériel, en particulier l'implantation du calcul correspondant à l'équation de récurrence, est à l'étude.

2.6.5 Perspectives

Dans l'avenir immédiat, nous comptons compléter ce générateur, et le tester afin de pouvoir le mettre prochainement à la disposition de la communauté. Une fonctionnalité intéressante que nous voulons implanter est la génération, en parallèle de la description VHDL d'opérateurs, de programmes logiciels se comportant comme ces opérateurs dans des buts de tests et de validation numérique.

CHAPITRE 3

Architectures partagées pour fonctions cryptographiques

Nous montrons ici qu'une architecture partagée pour des fonctions cryptographiques proches permet d'obtenir une performance comparable à des implantations séparées tout en demandant bien moins de matériel. Nous proposons une implantation améliorée des fonctions de hachage de la famille SHA-2, avec une latence des opérateurs minimale et des besoins matériels réduits. Nous proposons aussi une version fonctionnant à haute fréquence pour un coût de deux cycles de latence supplémentaires seulement par message. Finalement nous introduisons l'architecture multifonction capable d'effectuer soit un hachage SHA-384 ou SHA-512 ou de se comporter comme deux opérateurs SHA-224 ou SHA-256 indépendants.

En septembre et octobre 2005, dans le cadre d'une collaboration entre le LIP et le laboratoire ATIPS de l'Université de Calgary, j'ai eu l'occasion de travailler avec Ryan Glabb, alors en thèse à l'ATIPS. Ryan a travaillé un mois à Lyon, puis je suis parti à l'université de Calgary pendant le mois d'octobre.

Le but de notre collaboration était d'implanter efficacement plusieurs fonctions cryptographiques au sein d'un même composant, en partageant autant que possible le matériel afin d'en minimiser la taille.

Un candidat potentiel était la fonction de hachage WHIRLPOOL [64], dont le principe mathématique est proche de la norme de chiffrement AES [58]. Cependant les tables de substitution, qui occupent la majorité de la surface lors de l'implantation matérielle d'AES, ne sont pas à notre connaissance partageables entre AES et WHIRLPOOL.

Nous avons finalement choisi de travailler sur les fonctions de hachage de la famille SHA-2 [61] pour plusieurs raisons. Tout d'abord, la norme de hachage précédente SHA-1 [60] a été « cassée » [57]. La nouvelle norme de hachage SHA-2 n'a jusqu'ici bénéficié que de peu d'implantations matérielles. De plus toutes les fonctions définies par cette norme sont basées sur des algorithmes proches, ce qui a permis un fort partage de matériel.

Suite à cette collaboration, un article [6] a été présenté lors de la conférence internationale ERSA (*Engineering of Reconfigurable Systems & Algorithms*) à Las Vegas en 2006, et une version plus complète [7] est parue dans un numéro spécial intitulé « Embedded Cryptographic Hardware » du *Journal of System Architectures* en 2007.

3.1 Introduction

3.1.1 Fonctions de hachage cryptographique

Une fonction de hachage [59] prend en entrée un *message* de taille variable, et produit en sortie un *condensé* (en anglais *hash-value*). Plus formellement, une fonction de hachage h fait correspondre à des chaînes binaires de longueur arbitraire mais finie des chaînes de longueur n fixe. En général, chaque condensé est l'image d'un grand nombre de messages différents, puisqu'en supposant qu'on limite les entrées à une longueur maximale de t bits ($t \gg n$), et que h peut-être considérée comme aléatoire en ce sens que toutes les sorties sont équiprobables, alors chaque condensé représente environ 2^{t-n} messages différents.

Deux messages pris aléatoirement ont une probabilité 2^{-n} d'être hachés vers le même condensé. L'idée des fonctions de hachage cryptographique est de considérer le condensé comme un représentant plus court du message dont il est l'image, et de l'utiliser comme s'il identifiait ce message de manière unique. Un tel condensé est alors appelé une *empreinte* (en anglais *message digest*).

Les fonctions de hachage cryptographique sont un outil fondamental de la cryptographie moderne, utilisées principalement pour assurer l'intégrité des données lors de transmissions sur des canaux non sécurisés. Les fonctions de hachage sont aussi utilisées pour l'implantation d'algorithmes de signature numérique, des codes d'authentification de message et dans les générateurs de nombres aléatoires.

Une utilisation typique des fonctions de hachage pour assurer l'intégrité des données est la suivante. L'empreinte $h(x)$ d'un message x est calculée. L'intégrité de cette empreinte (pas celle du message lui-même) est protégée d'une manière quelconque. Plus tard, après un stockage sur un support sujet à dégradation, ou après avoir transféré le message sur un réseau non sécurisé, on veut savoir si le message récupéré x' correspond bien au message x original. On calcule son empreinte $h(x')$, que l'on compare à l'empreinte protégée $h(x)$. Si elles sont égales, on considère que les entrées x et x' sont bien identiques, donc x n'a pas été altéré. Le problème d'assurer l'intégrité d'un long message x est réduit à celui de l'intégrité de son empreinte $h(x)$ qui est bien plus courte, donc plus facile à protéger. Par exemple, c'est de cette façon que l'on s'assure de l'intégrité de l'image CD d'une distribution Linux. Il suffit à un utilisateur de calculer le condensé de l'image qu'il a téléchargé, habituellement avec la fonction MD5 [65], et de le comparer avec celui de l'original disponible sur internet.

Dans la mesure où le cardinal du domaine (les messages) de la fonction h est très supérieur à celui de son codomaine (les empreintes), l'existence de collisions (des messages différents partageant une même empreinte) est inévitable. Pourtant, un condensé devrait ne pouvoir être identifiée qu'à un seul message *en pratique*, c'est à dire que les collisions devraient être difficiles à trouver, voire impossible à calculer dans le cas des fonctions de hachage cryptographique.

Une fonction de hachage cryptographique h doit remplir les conditions suivantes :

Compression La fonction h fait correspondre à une entrée binaire x, de longueur arbitraire mais finie, une empreinte $h(x)$ de longueur n fixe.

Facilité de calcul Pour tout message x, l'empreinte $h(x)$ est facile à calculer.

Résistance aux pré-images Pour toute empreinte choisie à priori, le calcul d'un message qui se hache vers cette valeur n'est pas faisable. C'est à dire qu'on ne peut pas en un temps raisonnable calculer une pré-image x' telle que $h(x') = y$, en ayant pour donnée une empreinte y sans pré-image connue.

Résistance aux secondes pré-images Il n'est pas possible de calculer de message qui aie la même empreinte qu'un message donné, c'est à dire qu'on ne peut pas étant donné un message x trouver de second message x' qui vérifie $x' \neq x$ et $h(x) = h(x')$.

Résistance aux collisions Il n'est pas possible de calculer deux messages x et x' différents qui aient la même empreinte, c'est à dire qui vérifient $x' \neq x$ et $h(x) = h(x')$.

Un calcul que l'on qualifie de « facile » veut dire qu'il est possible d'obtenir rapidement un résultat, en pratique en quelques secondes de calcul sur ordinateur. Un calcul « impossible » veut dire qu'on ne peut espérer obtenir de résultat en un temps raisonnable, sauf progrès algorithmique imprévisible en l'état actuel des connaissances (par exemple dans la factorisation de grands entiers). Un seuil de robustesse généralement admis est de 2^{80} opérations nécessaires.

La résistance aux pré-images est liée à la fiabilité de certains protocoles de signature à clé publique, et les résistances aux secondes pré-images et aux collisions déterminent le niveau de sécurité des protocoles d'authentification et de vérification d'intégrité basés sur une fonction de hachage.

De nombreuses fonctions de hachage existent [65, 63, 62], mais leur véritable niveau de sécurité est très difficile à connaître. Lorsque des faiblesses sont découvertes [66], la sécurité des données est compromise et toutes les implantations matérielles et logicielles doivent être remplacées, ce qui entraîne des mises à niveau coûteuses vers de nouvelles fonctions de hachage estimées sûres pour le moment.

Par exemple, un algorithme récemment découvert [57] a permis de diminuer la résistance aux collisions de SHA-1 (Secure Hash Algorithm), la fonction de hachage la plus utilisée classiquement jusqu'ici. Le nombre d'opérations nécessaires pour générer des collisions s'en trouve ramené de 2^{80} à 2^{69}, ce qui place SHA-1 en deçà du seuil de sécurité estimé suffisant pour les opérations critiques. Depuis lors, la famille de fonctions de hachage SHA-2, développée par l'institut national des standards et technologies américain (NIST), est devenue la nouvelle norme.

À cause de leur complexité et de leur durée de vie limitée, les primitives cryptographiques sont généralement implantées en logiciel sur des processeurs généralistes, plutôt qu'en matériel sur support dédié. Une architecture matérielle est aussi en général beaucoup plus chère et difficile à réaliser de façon efficace. D'un autre coté, les algorithmes cryptographiques en logiciel sont bien moins rapides que leur contrepartie matérielle, typiquement plus lents par un facteur atteignant 1 à 3 ordres de grandeurs.

Une des raisons est que les primitives cryptographiques contiennent souvent des étapes qui requièrent un matériel spécialisé pour s'exécuter efficacement. Par exemple, une étape courante est la *diffusion* qui permute les bits d'un mot. En matériel une telle étape peut être réalisée par du routage, pour un coût et un délai presque nuls. En logiciel, il n'existe pas d'instruction adéquate [54]. Il va donc falloir sélectionner indépendamment chaque bit, puis le décaler et l'intégrer au résultat, ce qui est très lent et coûteux.

Nombre d'algorithmes cryptographiques tels qu'AES (Advanced Encryption Standard) et SHA-1 sont à l'origine crées en vue d'une implantation matérielle, et perdent leur efficacité de façon dramatique quand ils sont codés en logiciel [59]. En matériel, les deux principaux types de circuits pour l'implantation sont les circuits ASIC (Application-Specific Integrated Circuits) et les FPGA (Field-Programmable Gate Array). Les FPGA sont plus faciles d'utilisation et ont un faible coût d'exploitation. D'autre part, lors d'une mise à niveau vers une nouvelle norme cryptographique, ou pour une amélioration de l'algorithme, on n'aura besoin que d'appliquer un correctif logiciel pour une implantation à base de FPGAs, alors que dans le cas de circuits ASIC il faudra redessiner des masques et fondre de nouveaux circuits, pour un coût exorbitant. Notre choix s'est donc porté

Algorithme	**Tailles**				**Niveau de sécurité**
	mot (w)	**message** (l)	**bloc** (m)	**empreinte**	
SHA-224	32	$< 2^{64}$	512	224	112
SHA-256	32	$< 2^{64}$	512	256	128
SHA-384	64	$< 2^{128}$	1024	384	192
SHA-512	64	$< 2^{128}$	1024	512	256

Table 3.1 *Caractéristiques des algorithmes de hachage. Les tailles sont en bits.*

sur des FPGA, des familles Virtex et Spartan3 de Xilinx, pour les phases de prototypage et pour les résultats de synthèse.

3.2 Le Standard SHA-2

Au cours de ce chapitre, les notations utilisées sont celles trouvées dans les spécifications de la norme SHA-2 [61]. Cette norme détaille toutes les étapes des algorithmes et les constantes utilisées dans le calcul de l'empreinte. Ici en sont seulement reportées les parties utiles à la compréhension de l'implantation et des problèmes d'optimisation qui seront traités.

La norme de hachage SHA-2 comprend quatre algorithmes de hachage sécurisés : SHA-224 (qui calcule une empreinte sur 224 bits), SHA-256, SHA-384 et SHA-512. Ces quatre algorithmes sont basés sur une amélioration de la norme précédente SHA-1 [60]. Ce sont des fonctions de hachage cryptographique itératives qui traitent un message et produisent une empreinte de taille 224, 256, 384 ou 512 bits. Chaque algorithme procède en deux temps : un pré-traitement suivi du calcul de l'empreinte. Le pré-traitement demande tout d'abord de compléter le message par *remplissage* (*padding*), puis son découpage en *blocs* de m bits, et l'initialisation des variables utilisées au cours du hachage. Une suite de *sous-messages* (*message schedule*) est dérivée à partir de chaque bloc du message complet. Cette suite et une suite de constantes sont utilisées dans un calcul de ronde répété 64 à 80 fois pour obtenir itérativement une série de hachés intermédiaires. Le dernier haché, résultant du traitement du dernier bloc du message complété et du haché intermédiaire précédent, est utilisé pour déterminer l'empreinte.

Un message M de longueur l à hacher est traité par blocs de m bits. Chaque bloc est divisé en 16 mots de w bits pour le calcul. La longueur des mots w dépend de la fonction. On fait parfois référence aux fonctions de hachage SHA-224 et SHA-256, qui travaillent sur des mots de $w = 32$ bits, comme aux fonctions 32 bits de la norme, et aux fonctions SHA-384 et SHA-512 comme aux fonctions 64 bits. 32 ou 64 fait alors référence à la taille des mots w dans l'algorithme, et non à la taille de l'empreinte.

La principale différence entre les quatre algorithmes est la taille de l'empreinte. De plus, les algorithmes diffèrent en termes de taille de blocs et de mots. Ces tailles sont décrites dans le tableau 3.1. Pour un niveau de sécurité s, il faut 2^s opérations pour calculer une collision, en supposant que la fonction de hachage n'a pas de faille cachée.

3.2.1 Fonctions et constantes

Fonctions utilisées par les algorithmes

Chaque algorithme pour les fonctions de hachage utilise six fonctions logiques. Ces fonctions opèrent bit-à-bit sur des mots de w bits, notés x, y et z. Le résultat est lui aussi un mot de w bits. En plus de deux fonctions communes aux quatre algorithmes, $\text{Ch}(x, y, z)$ (fonction de choix) et $\text{Maj}(x, y, z)$ (fonction de majorité), quatre fonctions sont définies pour SHA-224 et SHA-256 ($w = 32$ bits) d'une part, SHA-384 et SHA-512 ($w = 64$ bits) d'autre part ; l'opération OU exclusif ($\oplus$) dans ces fonctions peut être remplacée par un OU ($\wedge$) et produira un résultat identique. Les fonctions $\text{ROR}_n(x)$ et $\text{SHR}_n(x)$ représentent respectivement la rotation vers la droite et le décalage vers la droite de n positions.

$$\text{Ch}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

$$\text{Maj}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

Les fonctions 32 bits utilisées pour SHA-224 et SHA-256 sont les suivantes :

$$\begin{aligned}
\Sigma_0^{\{256\}}(x) &= \text{ROR}_2\ (x) \oplus \text{ROR}_{13}(x) \oplus \text{ROR}_{22}(x) \\
\Sigma_1^{\{256\}}(x) &= \text{ROR}_6\ (x) \oplus \text{ROR}_{11}(x) \oplus \text{ROR}_{25}(x) \\
\sigma_0^{\{256\}}(x) &= \text{ROR}_7\ (x) \oplus \text{ROR}_{18}(x) \oplus \text{SHR}_3\ (x) \\
\sigma_1^{\{256\}}(x) &= \text{ROR}_{17}(x) \oplus \text{ROR}_{19}(x) \oplus \text{SHR}_{10}(x)
\end{aligned}$$

Les algorithmes SHA-384 et SHA-512 utilisent les fonctions 64 bits suivantes :

$$\begin{aligned}
\Sigma_0^{\{512\}}(x) &= \text{ROR}_{28}(x) \oplus \text{ROR}_{34}(x) \oplus \text{ROR}_{39}(x) \\
\Sigma_1^{\{512\}}(x) &= \text{ROR}_{14}(x) \oplus \text{ROR}_{18}(x) \oplus \text{ROR}_{41}(x) \\
\sigma_0^{\{512\}}(x) &= \text{ROR}_1\ (x) \oplus \text{ROR}_8\ (x) \oplus \text{SHR}_7\ (x) \\
\sigma_1^{\{512\}}(x) &= \text{ROR}_{19}(x) \oplus \text{ROR}_{61}(x) \oplus \text{SHR}_6\ (x)
\end{aligned}$$

Constantes de ronde

Les fonctions SHA-224 et SHA-256 utilisent soixante-quatre constantes de 32 bits, notées $K_0^{\{256\}}, K_1^{\{256\}}, \ldots, K_{63}^{\{256\}}$. Ces constantes sont en fait les parties fractionnaires des racines cubiques des 64 premiers nombres premiers.

Quatre-vingts constantes de 64 bits sont utilisées pour SHA-384 et SHA-512. Elles sont notées $K_0^{\{512\}}, K_1^{\{512\}}, \ldots, K_{79}^{\{512\}}$. Ces constantes représentent les 64 premiers bits des parties fractionnaires des racines cubiques des 80 premiers nombres premiers.

3.2.2 Pré-traitement

Cette opération consiste en trois étapes :
- Le remplissage du message M ;
- Le découpage du message complété en blocs de m bits ;
- L'initialisation de la valeur de départ du haché intermédiaire notée $H^{(0)}$.

Remplissage du message

Le rôle de la première étape est de s'assurer que la longueur totale du message qui sera traité soit un multiple de 512 ou 1024 bits.

Prenons un message M d'une longueur de l bits. Un bit '1' est concaténé à la fin, suivi de k bits '0'. Pour SHA-224 et SHA-256, k est le plus petit entier positif vérifiant $l + 1 + k \equiv 448 \pmod{512}$. Un bloc de 64 bits représentant la longueur l du message est ensuite rajouté à la fin. La longueur totale du message complété est bien un multiple de 512 bits.

Exemple 3.1 *Le message « abc » (en ASCII 8 bits) devient :*

$$\underbrace{01100001}_{a}\ \underbrace{01100010}_{b}\ \underbrace{01100011}_{c}\ 1\ \overbrace{0\cdots0}^{423}\ \overbrace{00\cdots0\underbrace{11000}_{l=24}}^{64}$$

Le remplissage suit le même principe pour SHA-384 et SHA-512. Un message complété de taille multiple de 1024 bits est obtenu cette fois avec k satisfaisant l'équation $l + 1 + k \equiv 896 \pmod{1024}$, et en utilisant une longueur représentée sur 128 bits.

La représentation de la longueur du message sur 64 et 128 bits explique la limite sur la longueur du message indiquée dans le tableau 3.1.

Découpage du message complété

Le message est découpé en N blocs de m bits, $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. Les m bits de chacun de ces blocs peuvent être exprimés comme 16 mots de w bits chacun, et ce pour tous les algorithmes de hachage. Les w premiers bits du $i$ème bloc sont notés $M_0^{(i)}$, les w bits suivants sont $M_1^{(i)}$, et ainsi de suite jusqu'à $M_{15}^{(i)}$.

Initialisation de la valeur du haché intermédiaire ($H^{(0)}$)

Avant que le hachage d'un message ne commence, la valeur $H^{(0)}$ est stockée. Elle consiste en huit mots de w bits «aléatoires».

SHA-224 : $H^{(0)}$ est obtenu en prenant les 33ème à 64ème bits de la partie fractionnaire des racines carrées du neuvième au seizième nombre premier.

SHA-256 : On a $H^{(0)}$ en prenant les 32 premiers bits de la partie fractionnaire des racines carrées des huit premiers nombres premiers.

SHA-384 : Les mots de $H^{(0)}$ sont constitués des 64 premiers bits de la partie fractionnaire des racines carrées du neuvième au seizième nombre premier.

SHA-512 : $H^{(0)}$ consiste en les 64 premiers bits de la partie fractionnaire des racines carrées des huit premiers nombres premiers.

3.2.3 Algorithmes de hachage sécurisés

Calcul du haché pour SHA-256 et SHA-512

La description suivante s'applique à la fois à SHA-256 et SHA-512, afin de souligner leurs nombreuses similitudes. On pose $Alg = 256$, $w = 32$ et $t_{\text{max}} = 63$ pour SHA-256, et $Alg = 512$, $w = 64$ et $t_{\text{max}} = 80$ si c'est SHA-512 qui nous intéresse.

Les deux algorithmes utilisent :
- une suite de $t_{\text{max}} + 1$ sous-messages de w bits ;
- huit variables $a, b, \ldots, h$ de w bits chacune ;
- une valeur de haché intermédiaire de huit mots de w bits.

Après complétion du pré-traitement, chaque bloc $M^{(1)}, \ldots, M^{(N)}$ est traité dans l'ordre. Les additions $(+)$ sont toutes effectuées modulo 2^w. $a \parallel b$ représente la concaténation des mots a et b.

Pour i allant de 1 à N :
- Préparer le sous-message :

$$W_t = \begin{cases} M_t^{(i)} & 0 \leq t \leq 15 \\ \\ \sigma_1^{\{Alg\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{Alg\}}(W_{t-15}) + W_{t-16} & 16 \leq t \leq t_{max} \end{cases}$$

- Initialiser les huit variables, a, b, c, d, e, f, g, et h, avec la $(i-1)$-ème valeur du haché :

$$\begin{aligned} a &= H_0^{(i-1)} \\ b &= H_1^{(i-1)} \\ &\vdots \\ h &= H_7^{(i-1)} \end{aligned}$$

- Pour t allant de 0 à t_{max} rondes, effectuer :

$$\begin{aligned} T_1 &= h + \Sigma_1^{\{Alg\}}(e) + \text{Ch}(e, f, g) + K_t^{\{Alg\}} + W_t \\ T_2 &= \Sigma_0^{\{Alg\}}(a) + \text{Maj}(a, b, c) \\ h &= g \\ g &= f \\ f &= e \\ e &= d + T_1 \\ d &= c \\ c &= b \\ b &= a \\ a &= T_1 + T_2 \end{aligned}$$

- Calculer le i-ème haché intermédiaire :

$$\begin{aligned} H_0^{(i)} &= a + H_0^{(i-1)} \\ H_1^{(i)} &= b + H_1^{(i-1)} \\ &\vdots \\ H_7^{(i)} &= h + H_7^{(i-1)} \end{aligned}$$

Après avoir répété ces étapes un total de N fois (après avoir traité $M^{(N)}$), l'empreinte de M est :

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)} \parallel H_7^{(N)}$$

SHA-224

Cet algorithme est presque identique à SHA-256, à la différence près qu'il utilise une valeur initiale $H_0^{(0)}$ du haché différente, et que seuls les 224 premiers bits du haché final sont utilisés pour l'empreinte :

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)} \parallel H_6^{(N)}$$

SHA-384

De même, SHA-384 se comporte presque identiquement à SHA-512, mais avec une valeur de $H^{(0)}$ différente et en ne gardant que les 384 bits les plus à gauche du haché final comme empreinte ;

$$H_0^{(N)} \parallel H_1^{(N)} \parallel H_2^{(N)} \parallel H_3^{(N)} \parallel H_4^{(N)} \parallel H_5^{(N)}$$

3.3 Implantation des fonctions de SHA-2

Dans cette section est décrite une implantation matérielle des fonctions de hachage de la famille SHA-2 avec une latence nulle ; c'est à dire qu'il se passe exactement $t_{max}+1 = 64$ (respectivement $t_{max} + 1 = 80$) cycles entre l'entrée du premier mot d'un bloc $M^{(i)}$ et la sortie du haché intermédiaire $H^{(i)}$ correspondant pour un algorithme SHA où $w = 32$ bits (respectivement $w = 64$ bits). Les données $M_0^{(i)}$ à $M_{15}^{(i)}$ du $i$ème bloc sont envoyées au cours des 16 premiers cycles du calcul.

Cette architecture minimise non seulement le surcoût en temps de calcul, mais est aussi très petite. La principale raison est qu'elle évite tout stockage inutile de données. Cependant, le débit y est pénalisé par la longueur du chemin critique. Il est ensuite possible d'obtenir des vitesses compétitives en pré-calculant une partie des données d'un cycle au cours du cycle précédent, ce qui permet d'obtenir un chemin critique plus court. Cette amélioration ne demande qu'une très petite quantité de matériel supplémentaire, et rajoute seulement deux cycles de latence par message complet.

3.3.1 Structure générale des opérateurs

L'architecture générale, telle que présentée dans la figure 3.1, est une représentation à haut niveau de la division des principaux blocs fonctionnels. Cette architecture peut être appliquée à tous les modes de hachage décrits dans ce chapitre. Les fonctions des principaux blocs sont les suivantes :

L'unité de contrôle gère tout les processus et opérations de l'opérateur. Son but est de coordonner l'arrivée de nouveaux messages et le transfert des blocs qui en sont extraits, et d'assurer la mise à jour au bon moment des variables et du haché intermédiaire.

L'unité de remplissage s'occupe du pré-traitement du message, et gère les morceaux de blocs $M_j^{(i)}$ du message à hacher.

L'unité d'expansion du message génère les sous-messages W_t utilisés à chaque ronde de calcul.

L'unité des constantes est une mémoire où sont lues les constantes K_t pour chaque ronde.

L'unité de calcul de ronde met à jour les variables a à h à partir de leur valeur précédente, de K_t et de W_t.

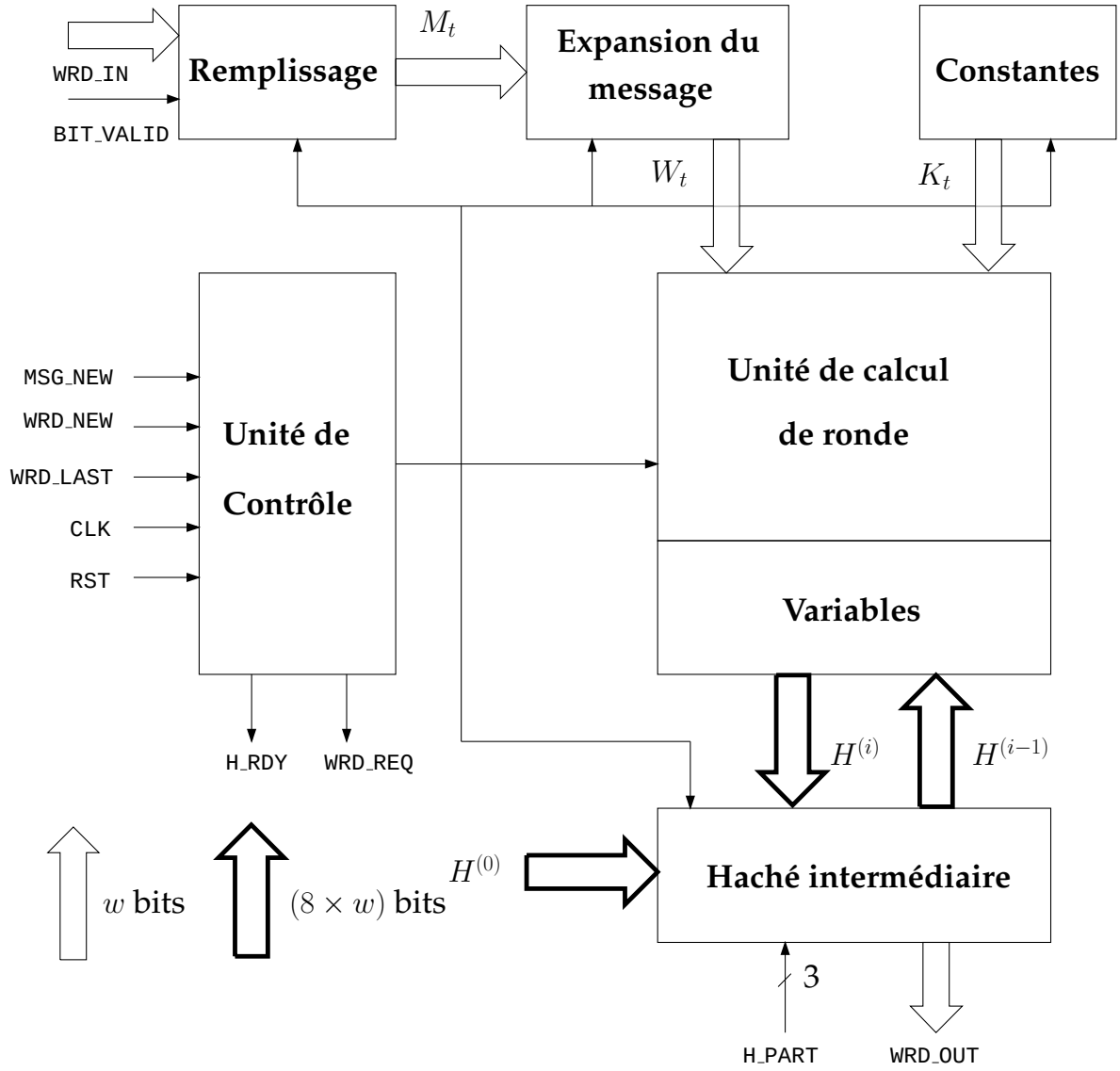

Figure 3.1 *Structure générale des opérateurs de la famille SHA-2.*

Le haché intermédiaire est initialisé à $H^{(0)}$ pour chaque nouveau message, et mis à jour à la fin du traitement de chaque bloc $M^{(i)}$ du message.

Le calcul réalisée par l'opérateur global commence lorsque la transmission d'un nouveau message à hacher est annoncée par le signal `MSG_NEW`. Le haché intermédiaire est alors initialisé à $H^{(0)}$. Au cours des seize premiers tours, M_t est calculé par l'unité de remplissage et envoyé à l'unité d'expansion du message qui transmet directement les premières valeurs de W_t. Ensuite, W_t est obtenu récursivement à partir des précédentes valeurs W_{t-2}, W_{t-7}, W_{t-15} et W_{t-16}.

En parallèle de W_t, la constante K_t est lue en mémoire à chaque ronde.

Les variables a à h sont quant à elles initialisées au début du traitement de chaque bloc avec la dernière valeur du haché intermédiaire $H^{(i-1)}$. Elles sont mises à jour 64 (respectivement 80) fois à l'aide de W_t et K_t. Ensuite, la nouvelle valeur du haché intermédiaire $H^{(i)}$ est produite par addition des variables a à h aux mots de $H^{(i-1)}$.

L'unité de remplissage reçoit ses entrées via le port `WRD_IN`. Lorsque le dernier bit du message arrive, le signal `WRD_LAST` l'indique, et le vecteur `BIT_VALID` indique le nombre de bits du

dernier mot transmis faisant effectivement partie du message. Le haché peut être lu sur le port WRD_OUT mot par mot en utilisant le port d'adresse H_PART.

3.3.2 Détail de l'implantation des unités

Dans cette première partie, les algorithmes de hachage SHA-2 sont implantés dans des composants séparés. Le but est dans un premier temps de minimiser le surcoût en temps de calcul, c'est à dire de mettre au point des composants capables de commencer à hacher un message aussitôt que possible.

Dès qu'assez d'information est disponible, le calcul doit démarrer sans attendre que la totalité du bloc (ou du message) soit arrivée. Bien que cette approche rende plus difficile l'écriture de composants se comportant correctement, elle a permis une forte réduction du coût matériel par rapport à d'autres architectures existantes. La principale raison est qu'en échange d'un contrôle plus compliqué, la mise en tampon de blocs du message est éliminée. La quantité de matériel dédié au contrôle s'accroît de façon négligeable, alors que le gain sur la mise en tampon diminue presque de moitié la surface des opérateurs.

Remplissage du message

Dans l'approche habituelle du problème le message est traité par blocs successifs en stockant un bloc complet dans un tampon, puis en le remplissant si nécessaire pour fournir des blocs complétés entiers à chaque fois. C'est une approche naturelle en logiciel où le stockage et la lecture de données sont triviales, par exemple lors de l'écriture de code destiné à être exécuté sur un processeur généraliste. Mais l'utilisation d'une stratégie de ce type en matériel implique la présence d'un registre de la taille d'un bloc, ainsi que des multiplexeurs permettant d'écrire et d'accéder aux mots de ce bloc indépendamment. Ceci introduit de plus une latence qui va atteindre 16 cycles si l'opérateur reçoit un nouveau mot du message à chaque cycle.

Cette latence peut être évitée, car à chaque ronde de calcul une seule valeur de W_t est requise. W_t est égal à M_t pour les 16 premières rondes, et ne dépend que de l'unité d'expansion du message au cours des rondes suivantes. L'unité de remplissage n'a donc à calculer M_t que pour les 16 premiers cycles du traitement de chaque bloc, et ce calcul peut très bien être fait au vol, c'est à dire au fur et à mesure que les blocs du mot sont disponibles.

L'architecture proposée pour l'unité de remplissage (figure 3.2) traite les mots de w bits un par un. La longueur du message est comptée à partir du premier mot, qui est fourni en même temps que le signal MSG_NEW. L'unité incrémente le compteur pour chaque mot du message jusqu'à la levée du signal WRD_LAST, qui indique l'arrivée du dernier bit du message (indexé par BIT_VALID). Un masque filtre les bits valides, puis ajoute un bit '1' au message, suivi par autant de bits '0' que nécessaire. Lorsque les 14 premiers mots du dernier bloc sont passés (on a ajouté k bits '0' au message), la partie gauche puis droite du compteur, qui contient alors la représentation binaire de la taille du message, est envoyée dans l'unité d'expansion du message. Enfin, le signal PAD_END est levé pour indiquer que le message complet a été traité.

L'unité de remplissage demande la suite du message via le signal WRD_REQ, lit 16 nouveaux mots sur le port WRD_IN, calcule les M_t et les envoie sur le port WRD_OUT. Elle attend ensuite 48 (respectivement 64) cycles pour que le reste de l'opérateur aie fini de traiter le bloc courant. Ensuite, si le message n'est pas terminé (WRD_LAST n'a pas été levé), l'unité requiert 16 nouveaux blocs, sinon elle attend un nouveau message. Si l'unité requiert de nouveaux mots qui n'arrivent pas (WRD_NEW n'est pas levé), l'opérateur complet suspend le calcul en attendant la suite du message.

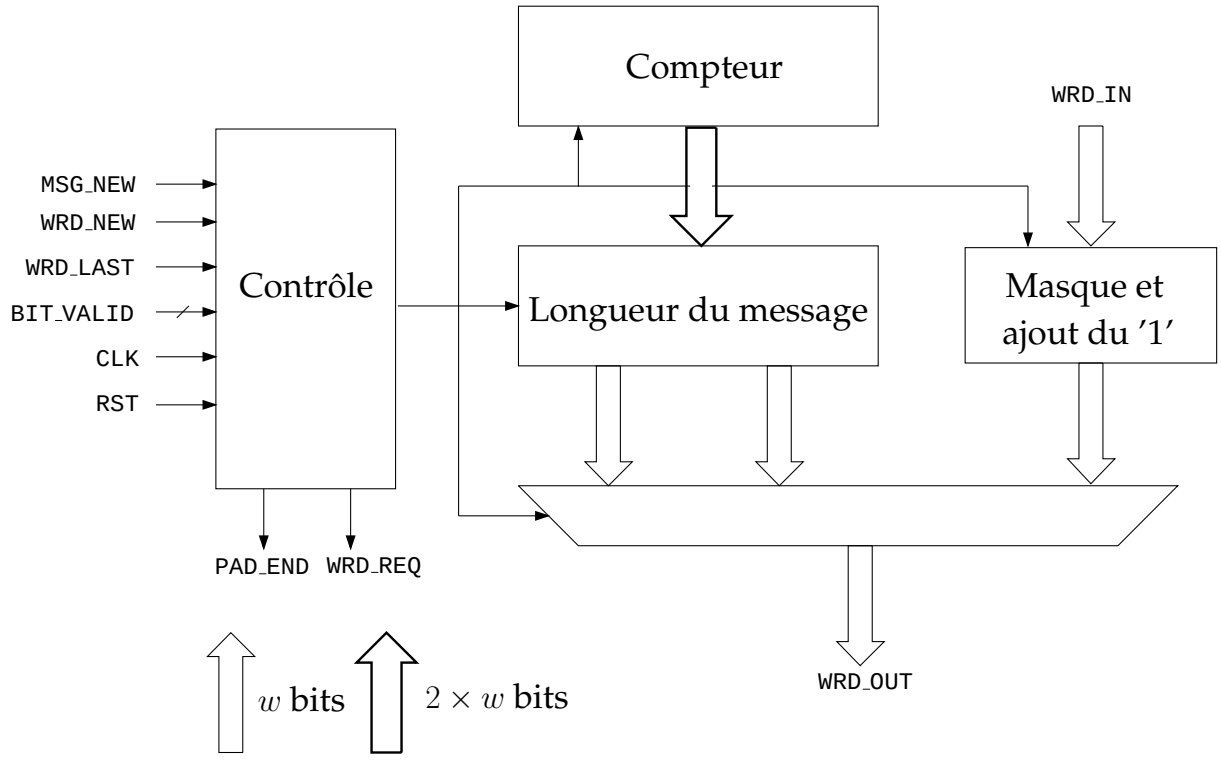

Figure 3.2 *Schéma d'implantation de l'unité de remplissage.*

Expansion du message

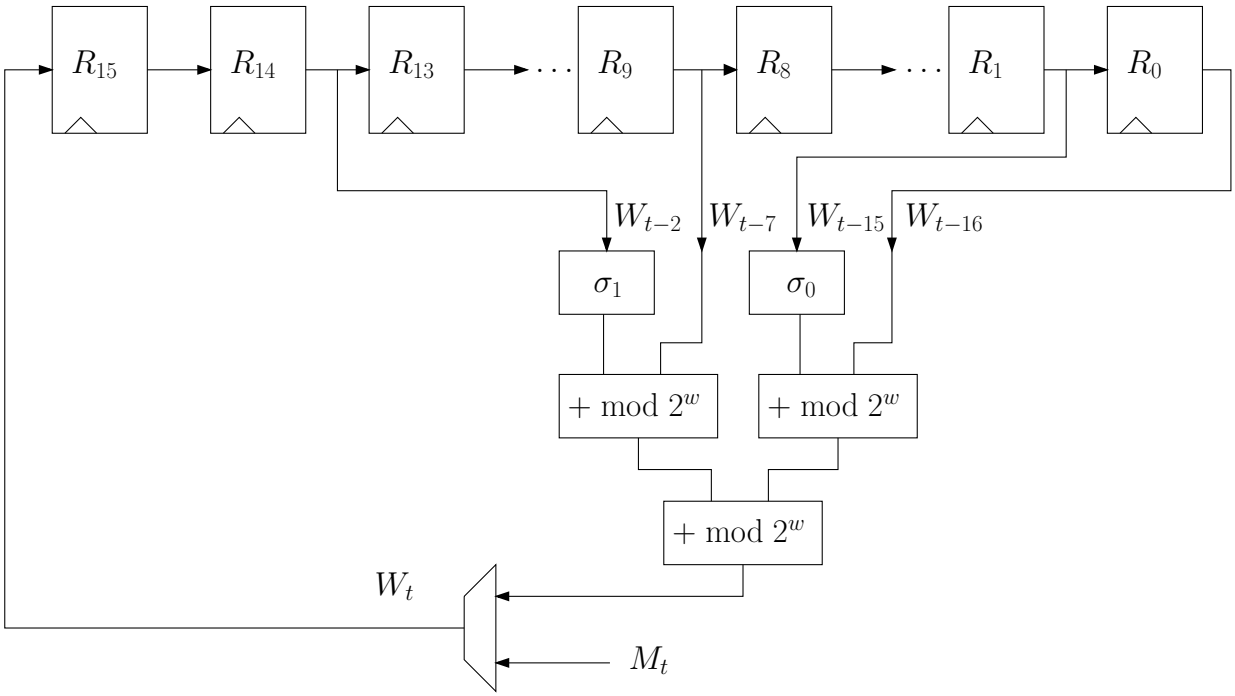

Figure 3.3 *Schéma d'implantation de l'unité d'expansion du message.*

De la norme, on sait que les mots W_0 à W_{15} ne sont pas calculés dans l'unité d'expansion,

t	$K_t^{\{256\}}$	$K_t^{\{512\}}$
0	428a2f98	428a2f98d728ae22
1	71374491	7137449123ef65cd
2	b5c0fbcf	b5c0fbcfec4d3b2f
3	e9b5dba5	e9b5dba58189dbbc

Table 3.2 *Quelques valeurs de K_t pour $w = 32$ et $w = 64$.*

puisqu'ils sont égaux aux mots M_0 à M_{15}. Il sont donc routés directement à travers l'unité d'expansion (voir figure 3.3). Seuls les W_t suivants sont calculés. Chacun dépend de 4 des 16 valeurs précédentes de W_t ($W_{t-2}, W_{t-7}, W_{t-15}$ et W_{t-16}). On doit donc stocker 16 mots contenant les précédents W_t pour être à même de former les nouveaux sous-messages. Pour cette raison W_t est réinjecté dans un pipe-line de 16 registres.

Constantes de ronde

Les constantes de ronde sont simplement lues à chaque ronde, l'unité peut être implantée dans une mémoire ROM. Un bloc de type RAM est utilisé à la place du fait de la cible de type FPGA. Chaque bloc RAM peut mémoriser 512 mots de 32 bits. Un seul de ces blocs est suffisant même pour les algorithmes SHA-384/512, car ces blocs sont capables de double port en lecture (deux mots stockés à des adresses différentes peuvent être lus à chaque cycle). Les constantes 64 bits de SHA-384 et SHA-512 sont mémorisées sur deux emplacements dont l'adresse diffère sur le premier bit. Elles sont lues en utilisant le double port, comme illustré dans la figure 3.4. La partie gauche des constantes 64 bits est lue sur le port A, adressé par $0t_6t_5 \dots t_0$, et sa partie droite sur le port B adressé par $1t_6t_5 \dots t_0$. Le port B n'est pas utilisé dans un opérateur SHA-224 ou SHA-256.

La latence de 1 cycle de ces blocs RAM est prise en compte par la partie contrôle de l'architecture.

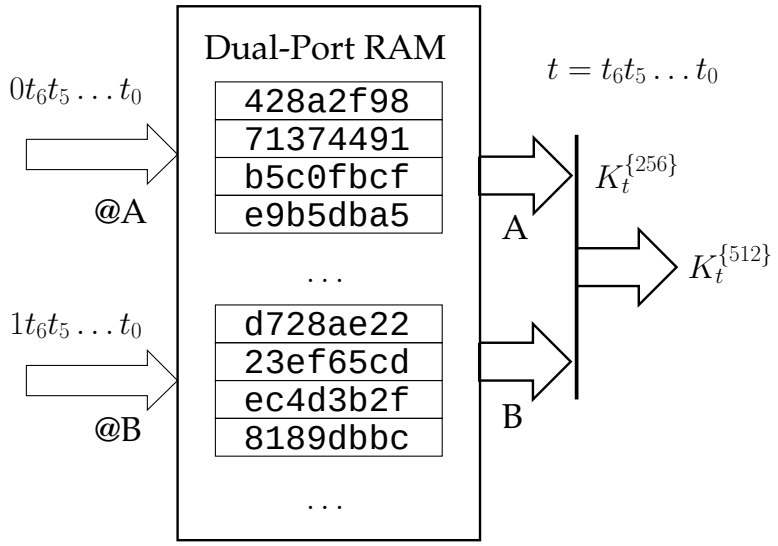

Figure 3.4 *Schéma d'implantation de l'unité de constantes.*

Calcul de ronde, variables $a, b, \ldots, h$

L'unité de calcul de ronde, étant donné le sous-message W_t, la constante K_t et les variables $a, b, \ldots, h$, calcule les valeurs suivantes des variables en appliquant les équations de la norme (figure 3.5). Ce calcul est effectué dans un arbre classique d'additionneurs à propagation de retenue, qui tirent partie des lignes de propagation rapide (fast-carry) disponibles sur les FPGA cibles.

Les variables sont initialisées au début d'un nouveau bloc de message par le haché intermédiaire $H^{(i-1)}$, et mises à jour à chaque ronde avec la sortie de l'unité de calcul de ronde.

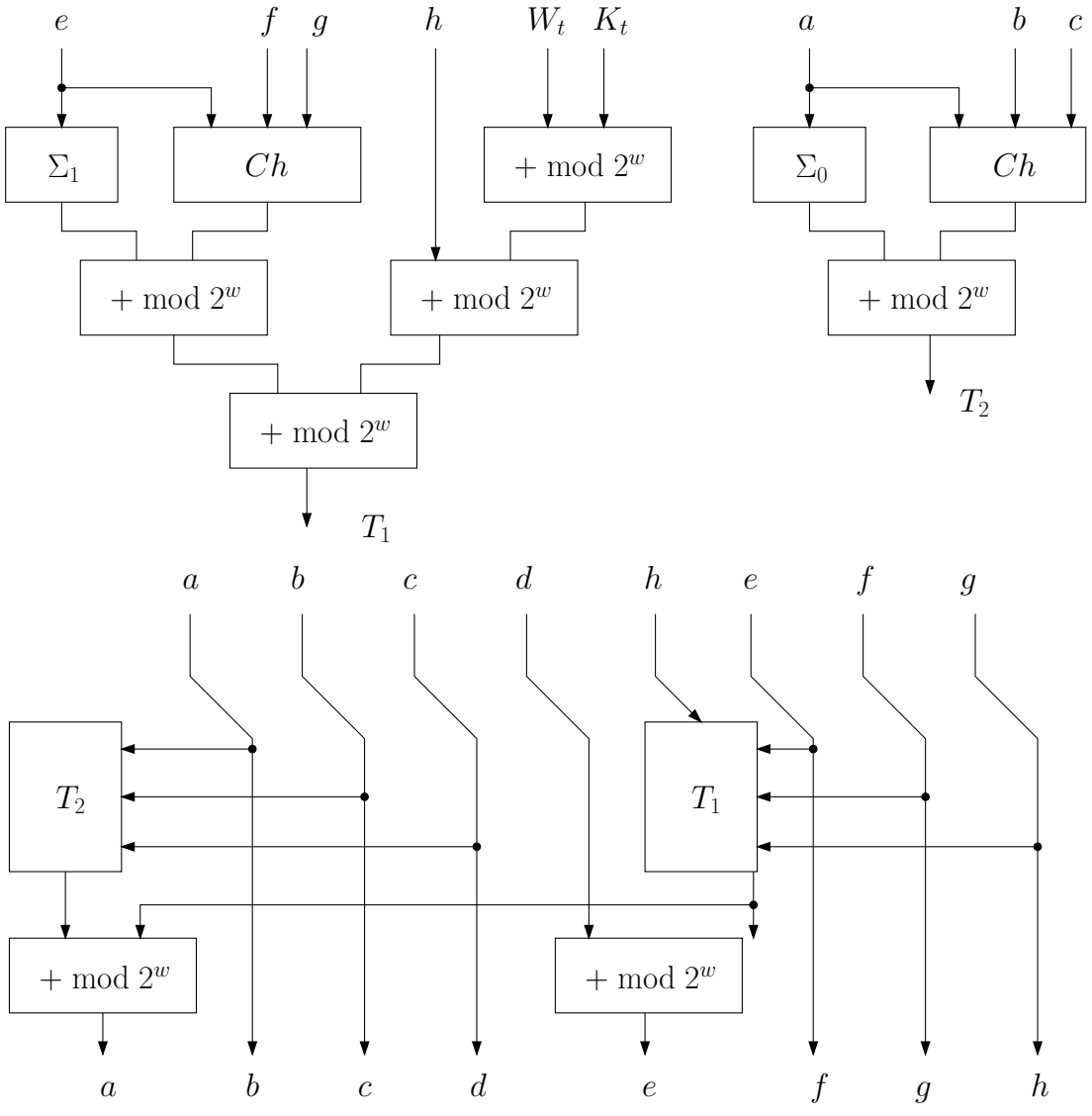

Figure 3.5 *Schéma d'implantation du calcul de ronde.*

Haché intermédiaire

Le haché intermédiaire est initialisé par $H^{(0)}$ à chaque nouveau message, et mis à jour par ajout des variables $a, b, \ldots, h$ sur chacun de ses mots après le traitement de chaque bloc.

3.3.3 Analyse des opérateurs

En calculant M_t et W_t au vol, l'opérateur est capable de démarrer un calcul dès l'arrivée du premier mot d'un message. Il est donc possible d'obtenir une latence nulle dans le calcul des hachés intermédiaires. Ainsi, pour un hachage SHA-224/256 (respectivement SHA-384/512), le traitement d'un message de N blocs prend $64 \times N$ (respectivement $80 \times N$) cycles exactement, en comprenant l'initialisation nécessaire au traitement d'un nouveau message.

L'autre avantage de cette approche est qu'en traitant toutes les données au fur et à mesure qu'elles arrivent, on se débarrasse du tampon habituellement utilisé pour la mémorisation des blocs du message. On réalise ainsi une économie importante de matériel par rapport à l'approche habituelle, où le calcul ne commence que lorsqu'un bloc entier est entré. La mise en tampon ajoute 16 cycles de latence par bloc, et nécessite un registre de la taille d'un bloc, plus le routage nécessaire à l'écriture et à la lecture de mots dans ce registre.

3.3.4 Fusion des opérateurs SHA-224/256, puis de SHA-384/512

On constate que la seule différence entre SHA-224 et SHA-256 repose dans la valeur de $H^{(0)}$ et dans le nombre de bits de haché qui sont utilisés.

Ces deux différences n'ont pas de conséquence sur le coût matériel : $H^{(0)}$ peut être considéré comme une séquence « aléatoire » de chiffres dans les deux cas, et le fait de ne lire qu'une partie du haché final pour l'empreinte ne permet pas d'économiser du matériel, puisque la totalité d'un haché intermédiaire est utilisée pour initialiser les variables avant le traitement du bloc suivant.

Le même raisonnement s'applique à SHA-384 et SHA-512 qui demandent pratiquement les mêmes ressources.

Deux composants supplémentaires, notés SHA-224/256 et SHA-384/512, ont été écrits. Ils possèdent une entrée supplémentaire ALT_HASH dont le rôle est de sélectionner quelle variante de l'algorithme utiliser (SHA-224 ou SHA-256 pour l'opérateur SHA-224/256, et SHA-384 ou SHA-512 pour l'opérateur SHA-384/512). Le seul matériel supplémentaire dont ces composants ont besoin par rapport aux implantations séparées des algorithmes est un multiplexeur utilisé pour router la bonne valeur de $H^{(0)}$.

3.4 Optimisation des opérateurs

La synthèse des opérateurs sus-décrits montre que le chemin critique est long, ce qui entraîne une vitesse de calcul faible, de l'ordre de 55 MHz pour SHA-512. Afin d'augmenter leur fréquence de fonctionnement, la ronde de calcul est modifiée pour répartir le chemin critique entre trois cycles. Les performances s'en trouvent accrues (64 MHz pour SHA-512), au prix de seulement deux cycles de latence par message et d'un peu de matériel supplémentaire.

3.4.1 Détermination du chemin critique

Les résultats de synthèse fournis par Synplify Pro montrent que le chemin critique traverse les unités de remplissage, d'expansion du message et de calcul de ronde (via l'évaluation de T_1), puis va à la mémorisation du haché intermédiaire (figure 3.6). Il est possible de réduire la longueur du chemin critique, et donc d'augmenter la fréquence de fonctionnement des opérateurs, au coût

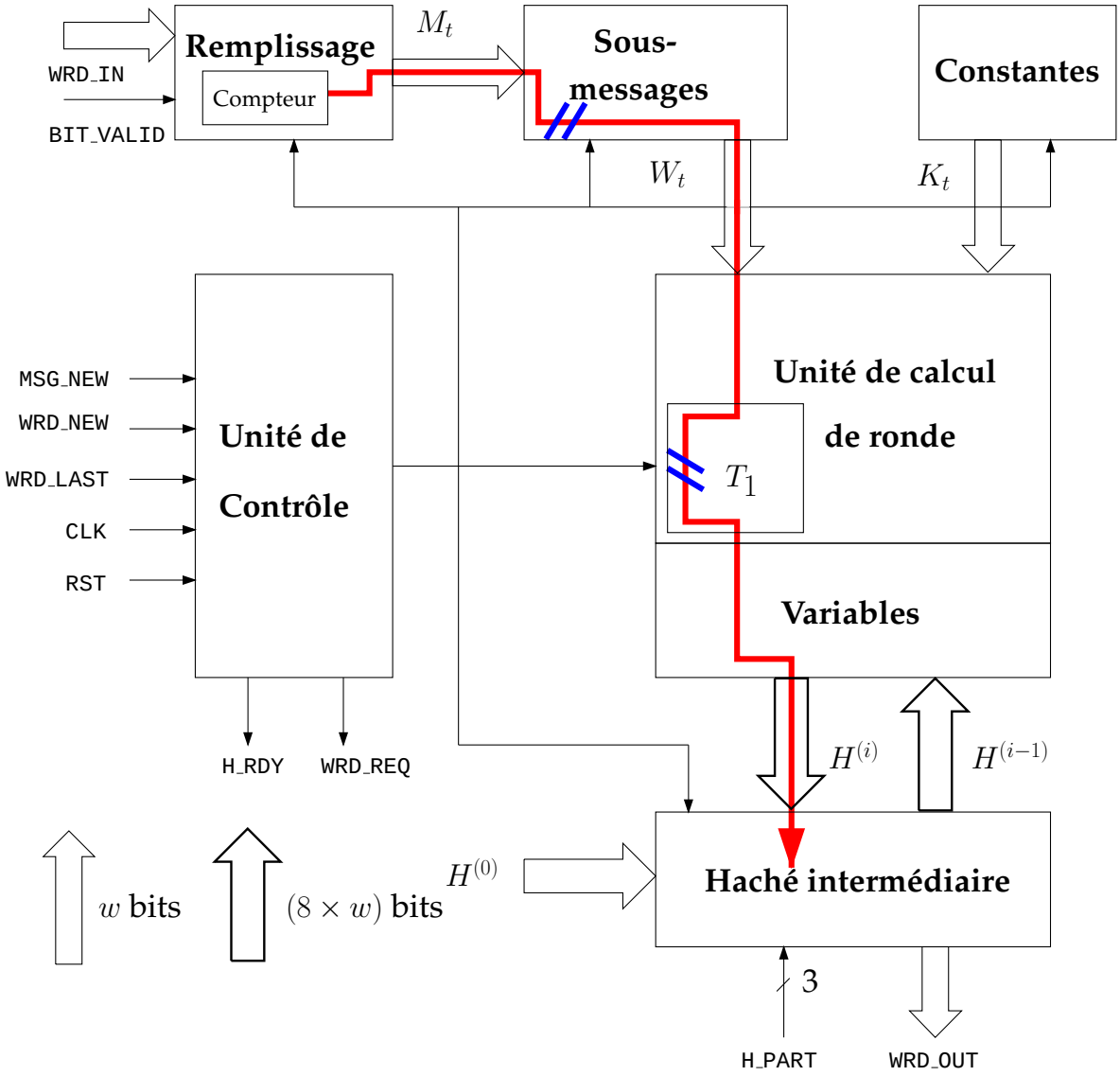

Figure 3.6 *Chemin critique dans l'implantation de SHA-2 (trait épais). On optimise en partageant le chemin critique entre trois cycles (barres).*

d'une petite augmentation de la latence. Pour ceci on va calculer à l'avance certaines des valeurs utilisées par le calcul de ronde au cours de la ronde précédente.

3.4.2 Découpage du chemin critique

Il est possible de réduire la longueur du chemin critique en calculant à l'avance certains résultats intermédiaires de la ronde de calcul. Par exemple, W_t peut être mémorisé dans un registre pour couper le chemin critique après les unités de remplissage et d'expansion du message. La latence est maintenant d'un cycle, car M_t n'est disponible pour le calcul qu'un cycle plus tard. Cette optimisation se fait sans ajout de registre, car W_{t-1} est de toute façon déjà mémorisé pour permettre l'expansion du message. Par contre K_t doit aussi être retardé d'un cycle, mais là encore on n'ajoute pas un registre de w bits, puisqu'il suffit de retarder l'adresse envoyée au bloc RAM. Cette adresse est égale au numéro de ronde, elle se mémorise dans un registre de 6 à 7 bits.

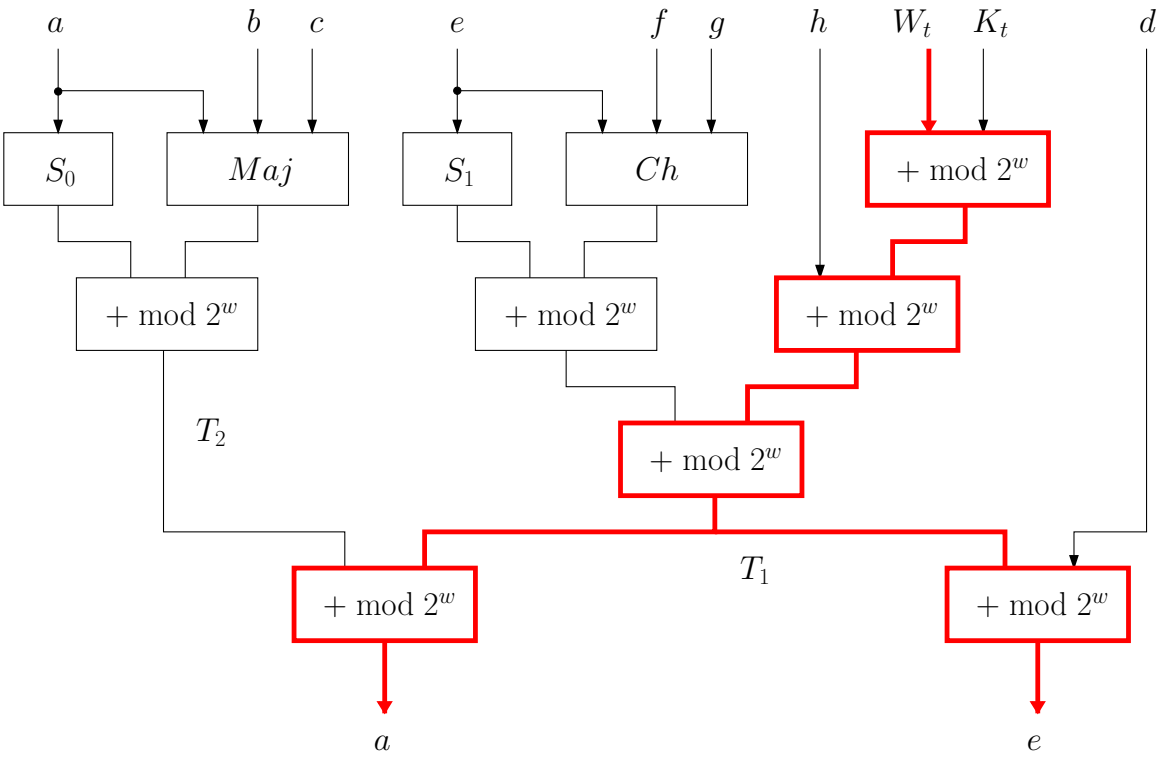

Figure 3.7 *Chemin critique à travers le calcul de ronde, avant optimisation (trait épais).*

En étudiant les équations qui régissent le calcul au cours d'une ronde, on constate que les variables c, d et g, h à l'itération t sont identiques (à part pour le premier tour quand $t = 0$) aux variables b, c et f, g de l'itération $t - 1$. Les pré-calculer est donc trivial. Il n'est par contre pas possible de pré-calculer efficacement a et e (respectivement b et f) car leurs valeurs dépendent de calculs sur les valeurs précédentes de a et e au temps t (respectivement $t - 1$).

Puisque le chemin critique passe par a et e (voir figure 3.7), on essaie de pré-calculer des expressions permettant d'accélérer leur évaluation. Les expressions $d + h + K_t + W_t$ pour e, et $h + K_t + W_t$ pour a peuvent être évaluées facilement dès la ronde $t - 1$. Il faut modifier les expressions utilisées pour le pré-calcul de h et d (c'est à dire g et c) afin de prendre en compte la ronde initiale $t = 0$ de hachage.

On introduit les nouvelles expressions :

$$c' = \begin{cases} H_3^{(i-1)} & \text{si } t = 0, \\ c & \text{sinon.} \end{cases}$$
$$g' = \begin{cases} H_7^{(i-1)} & \text{si } t = 0, \\ g & \text{sinon.} \end{cases}$$
$$p = g' + K_{t+1} + W_{t+1}$$
$$q = c' + g' + K_{t+1} + W_{t+1}$$
$$T_3 = \Sigma_1(e) + \text{Ch}(e, f, g)$$
$$T_4 = \Sigma_0(a) + \text{Maj}(a, b, c)$$

Il est maintenant possible de calculer rapidement a et e :

$$e = q + T_3$$
$$a = p + T_3 + T_4$$

Le chemin critique est désormais coupé après l'unité de remplissage et l'unité d'expansion du message : W_{t-1} est lu dans les registres de l'unité d'expansion à la place d'utiliser le résultat combinatoire W_t dans la ronde t. Il est aussi coupé dans l'unité de calcul de ronde : on pré-calcule p et q, qui sont utilisés avec T_3 et T_4 pour obtenir rapidement a et e (cf figure 3.8).

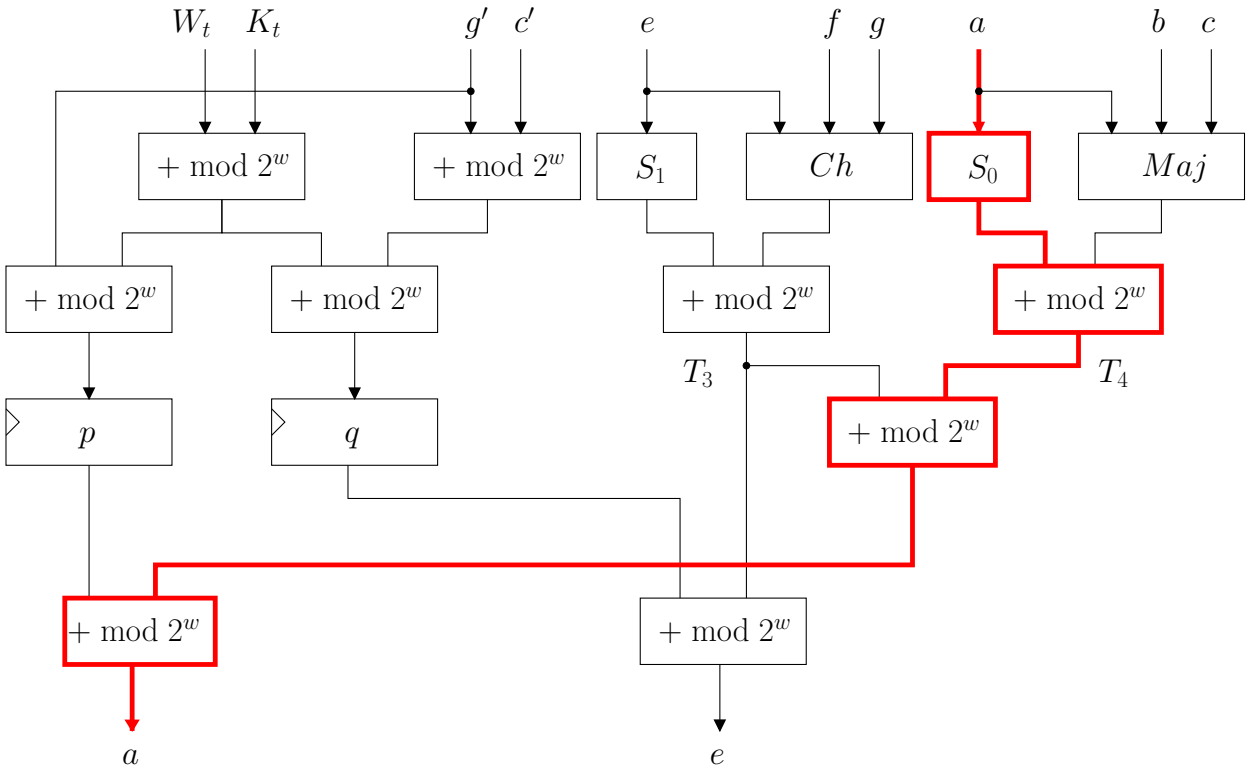

Figure 3.8 *Chemin critique après optimisation (trait épais).*

3.4.3 Analyse des opérateurs optimisés

La réduction du chemin critique a introduit deux cycles de latence : l'un est dû au fait de retarder W_t, et l'autre résulte du calcul des résultats intermédiaires p et q. Retarder W_t ne coûte aucun matériel, puisque W_{t-1} est mémorisé dans l'unité d'expansion du message. L'adresse t utilisée pour lire K_t doit être retardée (au coût de 7 flip-flops), et p et q sont mémorisés à chaque ronde de calcul (2 registres de w bits). Un peu de logique et de routage supplémentaires sont aussi nécessaires au calcul de c' et g'.

Le hachage d'un message de N blocs prend désormais $2 + 64 \times N$ (respectivement $2 + 80 \times N$) cycles pour SHA-224/256 (respectivement SHA-384/512). Le découpage du chemin critique, en plus d'augmenter la vitesse de fonctionnement des opérateurs, a une autre conséquence surprenante. Dans le cas d'un effort de synthèse en vitesse, le synthétiseur a beaucoup moins de difficultés à obtenir un opérateur efficace, car le chemin critique à optimiser est plus simple. Il impose donc moins de contraintes sur le placement. Le surcoût matériel lié au pré-calcul est alors plus que contrebalancé par la pression moindre le long du chemin critique. Raccourcir le chemin critique a permis la synthèse d'opérateurs plus petits ! La vitesse sur Spartan 3 augmente en moyenne de 7% pour des fonctions de hachage 32 bits, et de 17% pour des fonctions 64 bits. Le coût matériel est quant à lui *inférieur* de 15% pour 32 bits et 7% pour 64 bits.

3.5 Fusion des opérateurs de la famille SHA-2

Lorsqu'on veut calculer plusieurs fonctions de hachage, les fusionner dans une seule architecture est plus efficace que d'implanter des opérateurs séparés pour chaque algorithme. Dans [55], SHA-256, SHA-384 et SHA-512 sont chacun implantés dans un même opérateur, avec des unités de calcul de rondes séparées. Au cours d'un calcul avec la fonction SHA-256, cette architecture n'utilise que la moitié du chemin de données 64 bits, et force l'autre à zéro.

Nous avons conçus notre propre architecture multifonction pour SHA-2 à partir des implantations séparées. Cette unité a été étudiée pour optimiser l'utilisation du matériel. Elle est donc capable de calculer soit une fonction de hachage sur des mots de $w = 64$ bits (SHA-384 ou 512), ou deux fonctions où $w = 32$ bits (SHA-224 ou 256) fonctionnant de manière concurrente. Lors d'un calcul en mode concurrent, l'architecture peut être considérée comme deux opérateurs séparés calculant chacun une fonction de hachage 32 bits.

3.5.1 Comparaison entre les algorithmes

On veut fusionner les opérateurs en essayant de partager un maximum de matériel. Dans un premier temps on extrait autant de similitudes que possible entre les algorithmes.

Les fonctions de hachage de la famille SHA-2 sont très similaires. On peut les diviser en deux catégories : les fonctions où $w = 32$ bits, qui sont SHA-224 et SHA-256, et les fonctions où $w = 64$ bits, SHA-384 et SHA-512. Si on fait abstraction de la longueur w des mots, une grande partie du matériel est identique, et le reste peut être facilement partagé :

Le Remplissage est le même, à la longueur des mots près. Un message de longueur l est traité par blocs de 16 mots, un '1' lui est ajouté, suivi d'autant de zéros (k) que nécessaire pour que l'on aie $l + 1 + k \equiv 14w \pmod{(16w)}$. Une représentation binaire de l écrite sur deux mots est ensuite rajoutée.

L'expansion du message est identique dans toutes les fonctions, les fonctions σ mis à part. Ces fonctions ne dépendent que de la largeur des mots (un jeu de fonctions pour $w = 32$ bits, un autre pour $w = 64$ bits).

La valeur initiale du haché intermédiaire $H^{(0)}$ est différente pour tous les algorithmes. On peut cependant partager cette valeur entre les algorithmes de manière efficace. En effet, la moitié gauche des mots de $H^{(0)}$ pour SHA-512 sont les mots de $H^{(0)}$ pour SHA-256. De même pour SHA-384, les moitiés droites des mots de $H^{(0)}$ sont les mots de $H^{(0)}$ pour SHA-224.

Exemple 3.2 *Pour* $H_0^{(0)}$ *:*

Fonction	$H_0^{(0)}$
SHA-256	**6a09e667**
SHA-512	**6a09e667** f3bbc908
et	
SHA-224	**c1059ed8**
SHA-384	cbbb9d5d **c1059ed8**

Des fonctions définies par la norme SHA-2, seules les fonctions Ch et Maj sont identiques pour les quatre algorithmes. Les opérations σ et Σ sont différentes, bien qu'effectuant un traitement

similaire, c'est à dire un OU exclusif bit-à-bit de trois rotations/décalages de leur entrée. La valeur du décalage ou de la rotation diffère suivant la taille du mot, et le matériel ne peut donc pas être partagé. Puisque la valeur des rotations/décalages est fixe pour chaque algorithme, nous l'avons implantée sous la forme d'un simple routage. Il n'y a que deux jeux de fonctions σ et Σ à implanter (un pour $w = 32$ et l'autre pour $w = 64$). Les deux versions de chaque fonction sont donc implantées en parallèle, et le choix du résultat se fait par un multiplexeur. Le coût d'une telle structure est très inférieur à celui de l'opérateur générique basé sur des opérateurs de décalage/rotation à barillet que l'on trouve dans [55].

Les constantes de ronde sont les mêmes à taille de mot égale. De plus, la valeur de la constante K_t pour $w = 32$ est identique à la partie gauche de la constante K_t correspondante pour $w = 64$.

Exemple 3.3 *Pour $t = 0$:*

Fonction	K_0
SHA-224/256	**428a2f98**
SHA-384/512	**428a2f98** d728ae22

Le calcul de ronde et les définitions des variables sont les mêmes pour tous les algorithmes, bien que le nombre de rondes diffère suivant la taille du mot. On n'effectue que 64 rondes pour un hachage de type $w = 32$, et 80 pour un hachage $w = 64$.

3.5.2 Partage physique du matériel

L'architecture multifonction présentée fait tenir dans le même chemin de données deux calculs de fonctions de hachage pour des mots de 32 bits, ou un seul calcul opérant sur des mots de 64 bits. On note α et β les fonctions 32 bits qui utilisent respectivement les moitiés gauche et droite du chemin de donnée, et γ la fonction de hachage 64 bits qui en occupe la totalité.

Le partage physique est accompli en considérant toutes les opérations réalisées sur 64 bits par γ comme la juxtaposition de deux opérations 32 bits séparées pour α et β. Pour ce faire on va casser les dépendances entre les deux moitiés du chemin de données lors du calcul sur 2×32 bits.

Pour les registres et autres opérations intrinsèquement parallèles, par exemple les fonctions Ch et Maj, le partage physique n'implique aucun surcoût car les moitiés gauche et droite restent indépendantes quel que soit le mode d'opération (cf figure 3.9, en haut).

Par contre, pour d'autres opérations comme les additions modulo 2^w, il y a une propagation de retenue entre les parties gauche et droite dans l'addition modulo 2^{64} de γ qui doit être inhibée lors du calcul de deux additions parallèles modulo 2^{32} pour α et β (cf figure 3.9, en bas).

En plus du faible ajout de matériel pour les opérations, la partie contrôle doit être dupliquée afin de permettre à α et β de fonctionner de manière concurrente, et pas seulement en parallèle. Typiquement, l'unité de remplissage concurrente décrite plus loin devra être capable de traiter des blocs entrant pour α d'une part, et β d'autre part. Cette unité ne fonctionne que 16 cycles tous les 64 cycles, et il se peut que α et β ne soient pas en phase. De plus, tant qu'une requête pour un nouveau mot n'est pas satisfaite le calcul à travers l'opérateur est suspendu, il faut pouvoir figer indépendamment α et β.

La figure 3.9 montre les modifications requises sur un additionneur à propagation de retenue pour qu'il soit capable d'effectuer en utilisant le même matériel soit une addition modulo 2^{64} ou

deux additions modulo 2^{32} parallèles.

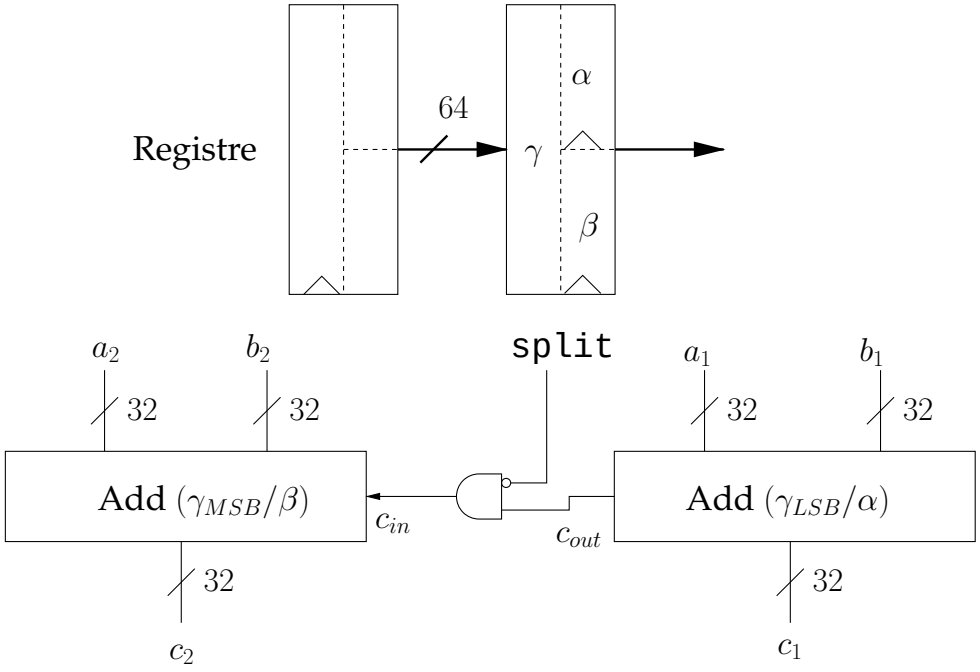

Figure 3.9 *Les registres peuvent être considérés soit comme un registre 64 bits ou comme deux registres 32 bits concurrents sans ajout de matériel, du fait de leur fonctionnement intrinsèquement parallèle (en haut). Additionneur à sélection modulo 2^{64} / 2×2^{32} (en bas).*

Remplissage du message

Dans la version multifonction de l'unité de remplissage, le compteur de mots est modifié pour qu'il puisse être utilisé soit comme un compteur 128 bits, soit comme deux compteurs 64 bits indépendants. Ceci implique un contrôle assez compliqué des retenues dans la mesure où les 4 (respectivement 5) derniers bits de chaque longueur de message pour un hachage $w = 32$ bits (respectivement $w = 64$ bits) sont transmis par l'entrée `BIT_VALID`. Si l'opérateur fonctionne en mode concurrent, la retenue propagée entre les deux moitiés du compteur 128 bits doit être ignorée, et `BIT_VALID` utilisé pour les bits de poids faible de la longueur du message β.

Expansion du message

L'expansion du message pour SHA-256 et SHA-512 est la même, à l'exception de :
- La longueur des mots qui varie du simple au double ;
- Les fonctions σ_0 et σ_1 qui doivent être implantées pour chaque mode, comme discuté plus haut ;
- Le nombre de rondes, qui n'affecte pas la combinatoire de l'unité (seulement le contrôle).

La figure 3.10 illustre la partie combinatoire du calcul multifonction de W_t. Des multiplexeurs routent le bon résultat pour chaque mode, et les additionneurs à sélection décrits plus haut sont utilisés pour les additions modulo 2^w.

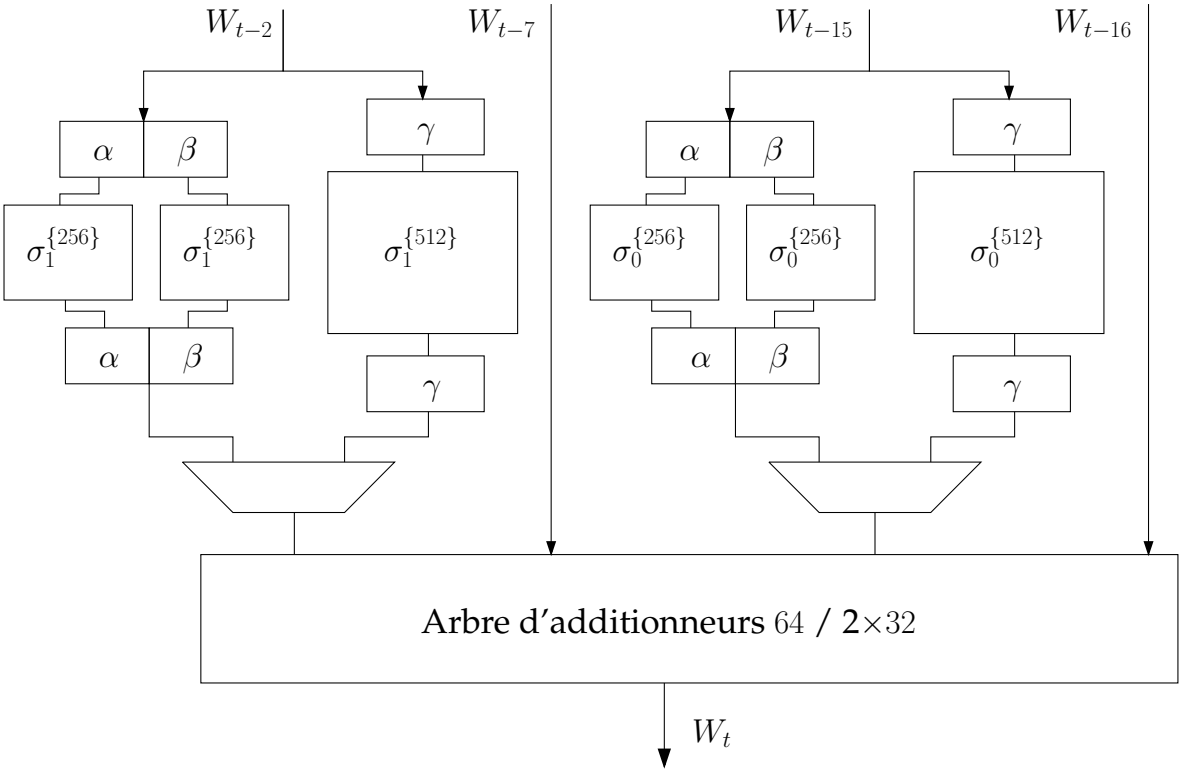

Figure 3.10 *Implantation multifonction du calcul du sous-message W_t.*

Constantes de ronde

Un seul bloc RAM 32 bits avec double port suffit pour obtenir les constantes de ronde 64 bits de SHA-384 et SHA-512 (voir figure 3.4). Le même bloc peut servir, moyennant un ajout de logique pour le contrôle, à fournir les constantes de deux fonctions de hachage 32 bits SHA-244/256 fonctionnant séparément. Les constantes $K_t^{\{512\}}$ bits de γ sont toujours lues sur A et B à la fois. Le port A est utilisé pour la constante $K_t^{\{256\}}$ bits de α et le port B sert à obtenir la constante $K_t^{\{256\}}$ pour β. Pour ce faire, le port B est maintenant adressé par $(\overline{\texttt{split}})\ t_6 t_5 \dots t_0$, où le signal `split` indique si l'architecture fonctionne en mode concurrent.

Pour assurer la même latence que dans les architectures séparées, il faut aussi ajouter un peu de logique pour assurer une initialisation correcte quand le mode de calcul vient de changer. En effet, l'unité de constantes doit alors fournir immédiatement soit la constante 64 bits $K_0^{\{512\}} =$ `428a2f98d728ae22`, soit deux constantes 32 bits $K_0^{\{256\}} =$ `428a2f98` en parallèle. Cependant, la lecture des constantes en mémoire introduit un cycle de latence. Pour résoudre ce problème, lors du cycle suivant un changement de mode de fonctionnement, la sortie du port B de la mémoire est modifiée par exclusion ($\oplus$) avec la valeur (`428a2f98` $\oplus$ `d728ae22`). Ainsi si la sortie du port B est la moitié droite de $K_0^{\{512\}}$ et que le mode devient concurrent, on obtient bien $K_0^{\{256\}}$, et inversement, et ce malgré la latence qu'impose le bloc mémoire.

Calcul de ronde, variables $a, b, \dots, h$

Les équations pour le calcul des nouvelles valeurs des variables $a, b, \dots, h$ sont les mêmes pour toutes les fonctions de la famille SHA-2, à la longueur des mots près et à l'exception des

fonctions Σ. La seule modification sur l'unité de calcul de ronde pour le multifonction va donc consister à utiliser des additionneurs à sélection de modulo et à implanter $\Sigma^{\{512\}}$ et $\Sigma^{\{256\}}$ pour chaque fonction Σ, à l'image de ce qui s'est fait pour l'unité d'expansion de messages 3.10.

Haché intermédiaire

La valeur initiale du haché intermédiaire $H^{(0)}$ est choisie à travers un circuit logique simple à base de multiplexeurs. Ce circuit utilise la similitude discutée plus haut de cette valeur pour les différents modes. La mise à jour du haché intermédiaire se fait en utilisant les additionneurs à sélection.

3.5.3 Analyse de l'architecture multi-mode

L'architecture multifonction partage les mêmes propriétés que les architectures séparées pour ce qui est de la latence et de la vitesse. La même amélioration que précédemment permet de diviser le chemin critique en trois pour accélérer la vitesse de fonctionnement en ajoutant deux cycles de latence par message.

3.6 Synthèse et résultats d'implantation

Cette section résume les résultats d'implantation obtenus pour les architectures présentées. Les synthèses sont effectuées par le logiciel Synplify Pro. Les critères étudiés sont l'occupation du FPGA en tranches, le débit maximum en Mbit/s et le ratio débit/occupation qui nous sert à caractériser l'efficacité de l'opérateur. Les opérateurs proposés ont étés synthétisés sur deux cibles : le Spartan3-XC3S400 et le Virtex 200/400XCV. La première cible a été choisie car elle contient des ressources en quantité suffisante pour y implanter tous les modes SHA-2, tout en étant d'une taille et d'un coût raisonnables. L'implantation sur Virtex a été effectuée à fins de comparaisons avec des architectures existantes. Le XCV200 est utilisé pour SHA-224 et SHA-256, et le XCV400 pour les fonctions 64 bits, qui demandent plus de ressources. Les simulations du VHDL sont faites en utilisant Modelsim.

Chaque fonction de hachage dans la famille SHA-2 a été synthétisée dans un opérateur séparé (224, 256, 384 et 512), puis fusionnée par taille de mot (224/256 ou 384/512). Les résultats pour l'architecture multifonction sont aussi donnés. Cette architecture est capable d'effectuer toutes les fonctions de la norme SHA-2, et peut se comporter comme deux opérateurs 32 bits (SHA-224/256) séparés opérant de manière concurrente.

Le choix d'un opérateur normal, fusionné ou multifonction est faite avant synthèse. Après implantation, des signaux de contrôle pilotent le mode de calcul.

Tous les opérateurs peuvent être synthétisés avec deux cycles de latence pour une fréquence accrue, ou sans latence mais avec un chemin critique plus long.

3.6.1 Vérification des opérateurs

La vérification des opérateurs s'est faite en deux parties. Dans un premier temps, un ensemble de tests implantant les vecteurs de tests spécifiés par la norme a été écrit. Leur hachage a été vérifié en comparant non seulement l'empreinte mais aussi les hachés intermédiaires et l'état des

variables avec leurs valeurs spécifiées par la norme. Dans un second temps, le comportement des composants a été testé intensivement par des simulations utilisant Modelsim.

La suite de test comprend un programme C qui génère aléatoirement des couples de test fonction/ message et les écrit dans un fichier, un second programme qui lit des triplets fonction/ message/ empreinte dans un fichier et les vérifie à l'aide des fonctions logicielles SHA-2 fournies par le NIST. Au milieu, un script Tcl pilote la simulation Modelsim d'un opérateur. Il lit les vecteurs de test fonction/message, et les envoie à l'opérateur en suivant le protocole décrit plus haut. Lorsque l'opérateur signale qu'il a fini un calcul, le script parcourt l'empreinte et écrit le vecteur fonction/message/empreinte dans le fichier réponse. En enchaînant les vecteurs de test, on va non seulement vérifier que les empreintes sont correctes mais aussi le protocole et le contrôle du composant. La fonction (224, 256, 384 ou 512) est tirée aléatoirement en fonction de l'opérateur à tester.

Dans le cas particulier de l'architecture multifonction, le script Tcl simule de plus un comportement d'ordonnanceur, et peut lancer plusieurs calculs SHA-224/256 de manière concurrente. Il a ainsi été possible de tester le comportement de cette architecture en mode concurrent, mais aussi au passage d'un mode à l'autre. C'est par exemple lors de ces tests qu'est apparu le problème de l'unité de constantes au passage d'un mode 64 bits à un mode 32 bits, résolu plus tard par modification de la sortie B avec la valeur $(\texttt{428a2f98} \oplus \texttt{d728ae22})$.

Tous les opérateurs ont été testés avec les vecteurs de la norme, puis avec plusieurs milliers de messages aléatoires enchaînés, en alternant les modes pour les opérateurs fusionnés. La suspension puis la bonne reprise du calcul en cas de famine a aussi été testée.

3.6.2 Résultats de synthèse

Tous les opérateurs ont été synthétisés d'une part en plaçant un effort sur la minimisation de la surface occupée, d'autre part en maximisant la vitesse des opérateurs.

L'efficacité d'une architecture est d'abord évaluée en termes de surface, mesurée en tranches (ou *slices*) du FGPA, ou en blocs logiques combinatoires (CLB). Le second critère est le débit maximum de l'opérateur, en mégabits par seconde. Enfin le rapport débit par surface, évalué en Mbit par seconde et par tranche, permet d'évaluer l'efficacité des architectures.

En termes de surface occupée, on s'attend à obtenir des résultats similaires pour les opérateurs séparés SHA-224, SHA-256 et l'opérateur fusionné SHA-224/256. En effet, les algorithmes sont quasiment identiques, et opèrent sur des mots de la même taille. De même, on s'attend à avoir des résultats proches pour les opérateurs SHA-384, SHA-512 et SHA-384/512. L'architecture multifonction pour la famille SHA-2, quant à elle, devrait occuper une surface légèrement supérieure à celle de SHA-512. Une partie du matériel ne peut pas être partagée entre tous les modes, et la majorité du contrôle est dupliquée pour permettre l'exécution concurrente de deux fonctions de hachage 32 bits. Cependant, on s'attend à ce que la flexibilité accrue, en particulier la possibilité de doubler le débit lorsqu'on travaille sur SHA-224/256, fasse plus que contrebalancer le coût matériel supérieur.

En termes de débit, les modes d'opération 32 bits (SHA-224 et SHA-256) sont capables de fournir une empreinte de 224 ou 256 bits tous les 64 tours, tandis que les modes d'opérations 64 bits calculent une empreinte de 384 ou 512 bits tous les 80 cycles, soit en 20% de temps en plus à fréquence d'horloge égale. De plus, le chemin critique à travers un chemin de données 64 bits est forcément plus long, puisqu'il comprend par exemple des additions à propagation de retenue. On peut donc s'attendre à une fréquence d'horloge réduite pour les modes 64 bits, donc une diminution

Opérateur	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
SHA-224	1488	65	64	227,5	0,153
SHA-256	1492	65	64	260	0,174
SHA-224/256	1509	64	64	224/256	0,170
SHA-384	2781	55	80	264	0,095
SHA-512	2781	55	80	352	0,127
SHA-384/512	2781	53	80	254,4/339,2	0,122
SHA-2 Multifonction	3245	45	64	2×180/288	0,111/0,089

Table 3.3 *Résultats de synthèse avec effort en vitesse pour Spartan 3 sans optimisation.*

Opérateur	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
SHA-224	1272	70	64	245	0,193
SHA-256	1306	70	64	280	0,214
SHA-224/256	1231	67	64	234,5/268	0,218
SHA-384	2609	64	80	307,2	0,118
SHA-512	2545	64	80	409,6	0,161
SHA-384/512	2485	62	80	297,6/396,8	0,160
SHA-2 Multifonction	2957	55	64	2×220/352	0.149/0,119

Table 3.4 *Résultats de synthèse avec effort en vitesse pour Spartan 3 avec optimisation.*

relative du débit au lieu d'une augmentation linéaire en la taille de l'empreinte.

3.6.3 Comparaisons entre les fonctions de hachage

Les résultats de synthèse sur Spartan 3, avec effort en surface, pour des opérateurs de latence minimale (donc avec un long chemin critique) montrent que les opérateurs SHA-224, SHA-256 et SHA-224/256 occupent des surfaces très proches avec respectivement 1488, 1492 et 1509 tranches. Les légères variations peuvent être imputées en partie au matériel pour la sélection de $H^{(0)}$ dans l'opérateur SHA-224/256, mais aussi aux optimisations effectuées par le synthétiseur. Les surfaces pour des opérateurs SHA-384, SHA-512 et SHA-384/512 sont identiques, et occupent 2781 tranches. Les tailles quasiment identiques pour les opérateurs 32 bits d'une part, et 64 bits d'autre

Opérateur	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
SHA-224	1321	68	64	238	0,180
SHA-256	1310	68	64	272	0,208
SHA-224/256	1338	66	64	231/264	0,197
SHA-384	2518	65	80	312	0,124
SHA-512	2524	65	80	416	0,165
SHA-384/512	2710	62	80	297.6/396,8	0,146
SHA-2 Multifonction	3330	46	64	2 ×184/294.4	0.111/0,088

Table 3.5 *Résultats de synthèse avec effort en vitesse pour Virtex sans optimisation.*

Opérateur	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
SHA-224	1297	77	64	269,5	0,208
SHA-256	1306	77	64	308	0,236
SHA-224/256	1260	69	64	241.5/276	0,219
SHA-384	2581	69	80	331,2	0,128
SHA-512	2545	69	80	441,6	0,174
SHA-384/512	2573	66	80	316.8/422,4	0,164
SHA-2 Multifonction	2951	50	64	2×200/320	0.136/0,108

Table 3.6 *Résultats de synthèse avec effort en vitesse pour Virtex avec optimisation.*

Opérateur	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
SHA-224	1125	50	64	175	0,156
SHA-256	1123	50	64	200	0,178
SHA-224/256	1147	50	64	175/200	0,174
SHA-384	2297	46	80	220,8	0,096
SHA-512	2297	46	80	294,4	0,128
SHA-384/512	2319	46	80	220.8/294,4	0,127
SHA-2 Multifonction	2687	38	64	2×152/243.2	0,113/0,091

Table 3.7 *Résultats de synthèse avec effort en surface pour Spartan 3 sans optimisation.*

Opérateur	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
SHA-224	1099	59	64	206,5	0,188
SHA-256	1098	59	64	236	0,215
SHA-224/256	1110	58	64	203/232	0,209
SHA-384	2210	50	80	240	0,109
SHA-512	2210	50	80	320	0,145
SHA-384/512	2172	50	80	240/320	0,147
SHA-2 Multifonction	2853	46	64	2×184/294.4	0.129/0,103

Table 3.8 *Résultats de synthèse avec effort en surface pour Spartan 3 avec optimisation.*

Opérateur	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
SHA-224	1111	48	64	168	0,151
SHA-256	1110	48	64	192	0,173
SHA-224/256	1120	47	64	164.5/188	0,168
SHA-384	2288	44	80	211,2	0,092
SHA-512	2288	44	80	281,6	0,123
SHA-384/512	2350	43	80	206.4/275,2	0,117
SHA-2 Multifonction	2562	36	64	2×144/230.4	0.112/0,090

Table 3.9 *Résultats de synthèse avec effort en surface pour Virtex sans optimisation.*

Opérateur	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
SHA-224	1159	56	64	196	0,169
SHA-256	1158	56	64	224	0,193
SHA-224/256	1108	53	64	185.5/212	0,191
SHA-384	2382	48	80	230,4	0,097
SHA-512	2384	48	80	307,2	0,129
SHA-384/512	2281	49	80	235.2/313,6	0,137
SHA-2 Multifonction	2464	42	64	2×168/268.8	0.136/0,109

Table 3.10 *Résultats de synthèse avec effort en surface pour Virtex avec optimisation.*

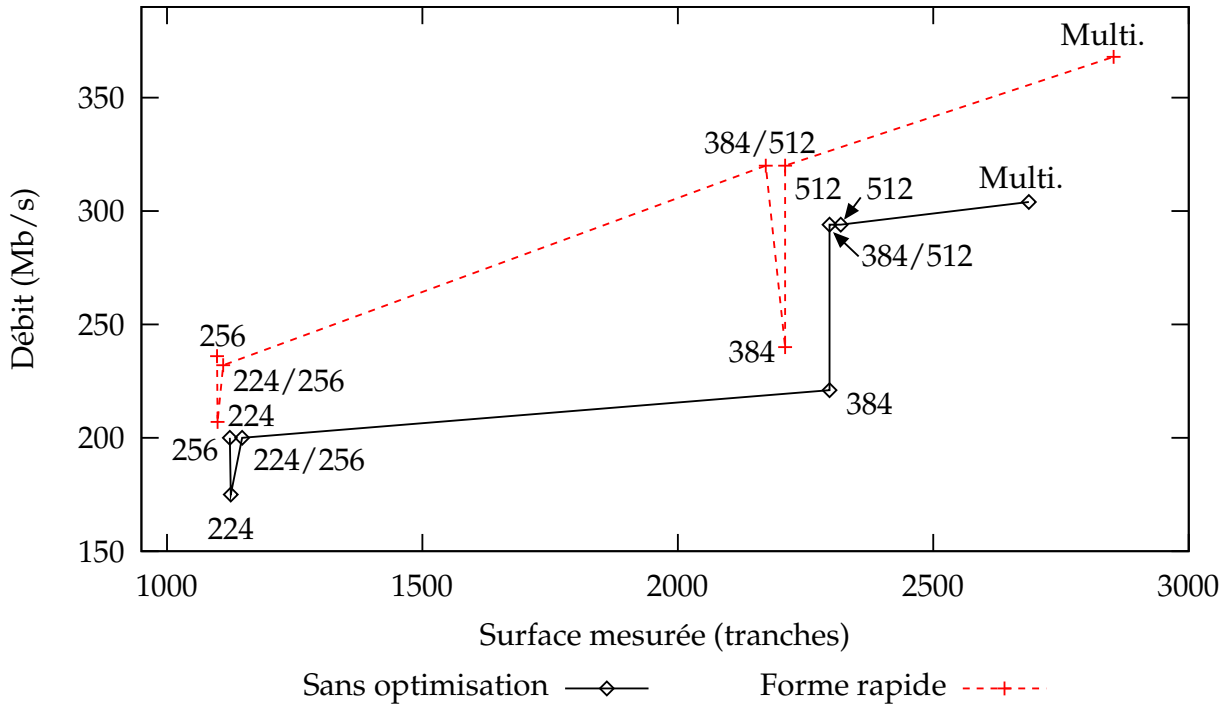

Figure 3.11 *Graphe débit/surface pour Spartan avec effort en surface. Toutes les architectures sont présentées sous leur forme normale (ligne continue) et sous leur forme rapide (en pointillés). Multi désigne l'architecture multifonction.*

part, sont particulièrement apparentes en synthèse avec effort sur la surface. Ce résultat correspond bien à ce qu'on attendait lors de l'étude des algorithmes. Il montre aussi que les modes alternatifs de calcul (SHA-224 et SHA-384) demandent les mêmes ressources que les modes principaux. La fusion des modes principaux et alternatifs dans les opérateurs SHA-224/256 et SHA-384/512 n'accroît pas le coût matériel de façon notable pour un effort en surface. Pour un effort en vitesse, les versions fusionnées occupent parfois moins de place.

Analyse des ressources utilisées

La comparaison des implantations avec latence minimale ou avec un latence de deux cycles fait apparaître plusieurs résultats intéressants. L'optimisation proposée devrait théoriquement accélérer la fréquence de fonctionnement des opérateurs en découpant le chemin critique, mais au prix d'une

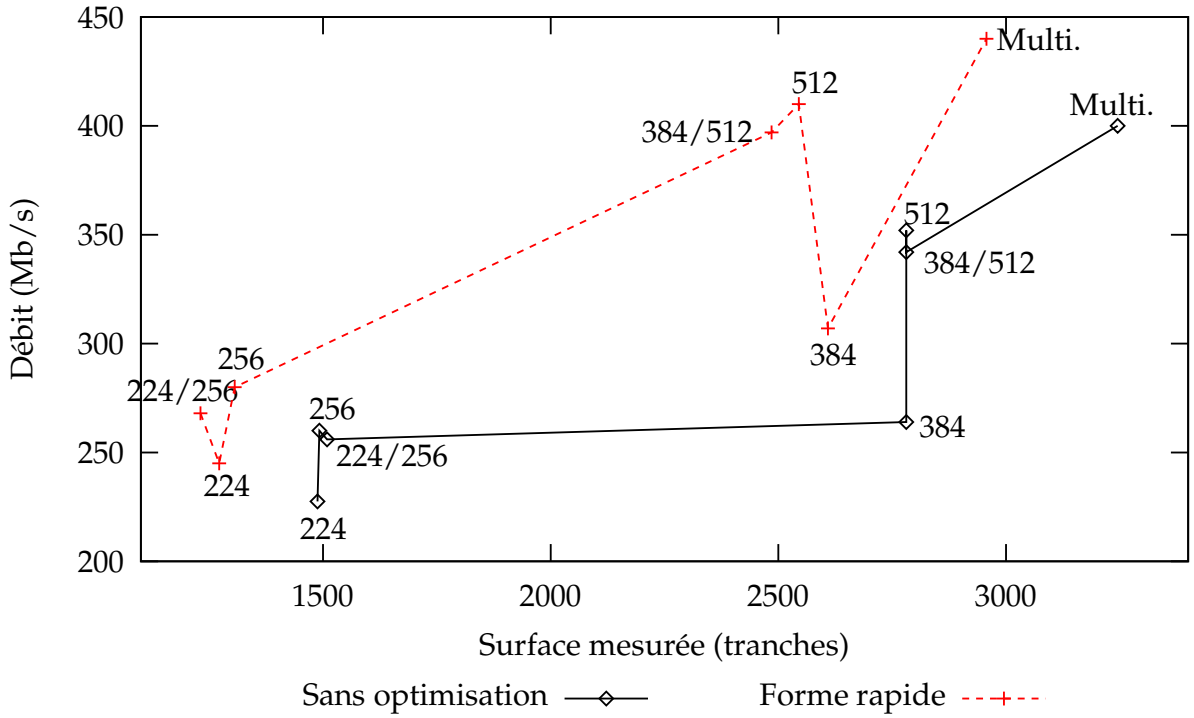

Figure 3.12 *Graphe débit/surface pour Spartan avec effort en vitesse. Toutes les architectures sont présentées sous leur forme normale (ligne continue) et sous leur forme rapide (en pointillés).*

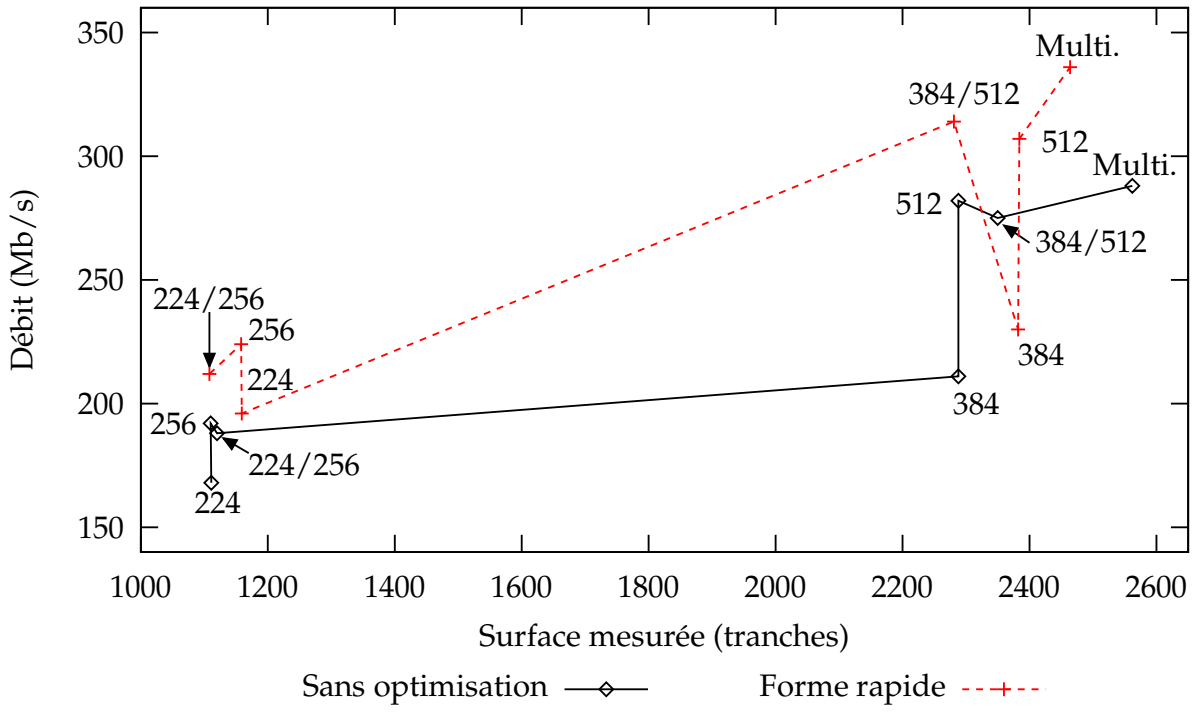

Figure 3.13 *Graphe débit/surface pour Virtex avec effort en surface. Toutes les architectures sont présentées sous leur forme normale (ligne continue) et sous leur forme rapide (en pointillés). Multi désigne l'architecture multifonction.*

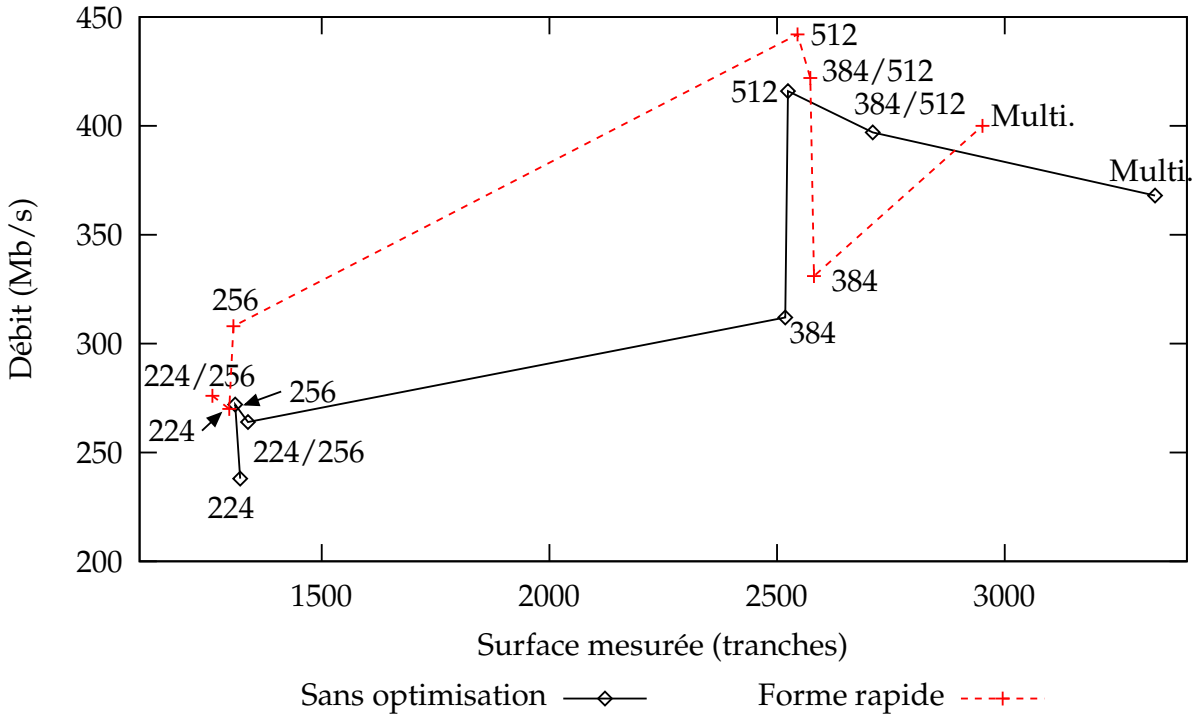

Figure 3.14 *Graphe débit/surface pour Virtex avec effort en vitesse. Toutes les architectures sont présentées sous leur forme normale (ligne continue) et sous leur forme rapide (en pointillés).*

petite quantité de matériel supplémentaire. Cependant les résultats pour en effort de synthèse en vitesse montrent au contraire une diminution du nombre de tranches occupés. En moyenne, les ressources après optimisation représentent 85% de celles avant modification pour les fonctions 32 bits, 93% pour les fonctions 64 bits et 91% dans le cas de l'opérateur multifonction.

La raison derrière cette diminution est que bien qu'on aie ajouté de la logique pour permettre le pré-calcul de certaines valeurs, la réduction de la longueur du chemin critique facilite le travail du synthétiseur. Les nouveaux chemins critiques sont mieux équilibrés, il n'y a plus un seul long chemin à privilégier, et moins de logique redondante est utilisée. Le placement des ressources en est facilité et la surface occupée diminue.

La diminution des ressources occupées est moins claire pour un effort de synthèse en surface. L'opérateur multifonction devient alors plus coûteux sous sa version rapide (4% de tranches occupés en plus). Ceci est dû au fait que le synthétiseur, cherche moins à optimiser le chemin critique, la surface dépend donc plus directement de la quantité de logique. Le matériel utilisé pour l'optimisation proposée transparaît alors dans le résultat.

L'effort imposé au synthétiseur a un effet important sur la surface des opérateurs. En moyenne un opérateur synthétisé avec un effort en surface occupe 86% des ressources dédiées au même opérateur synthétisé pour la vitesse.

Analyse de la fréquence de calcul

Pour les opérateurs présentés, deux facteurs influent fortement sur la vitesse de calcul. D'abord l'effort demandé au synthétiseur (surface ou vitesse). La fréquence d'opération varie presque de moitié suivant l'effort demandé au synthétiseur.

L'amélioration proposée diminue la longueur du chemin critique, ajoute une faible quantité

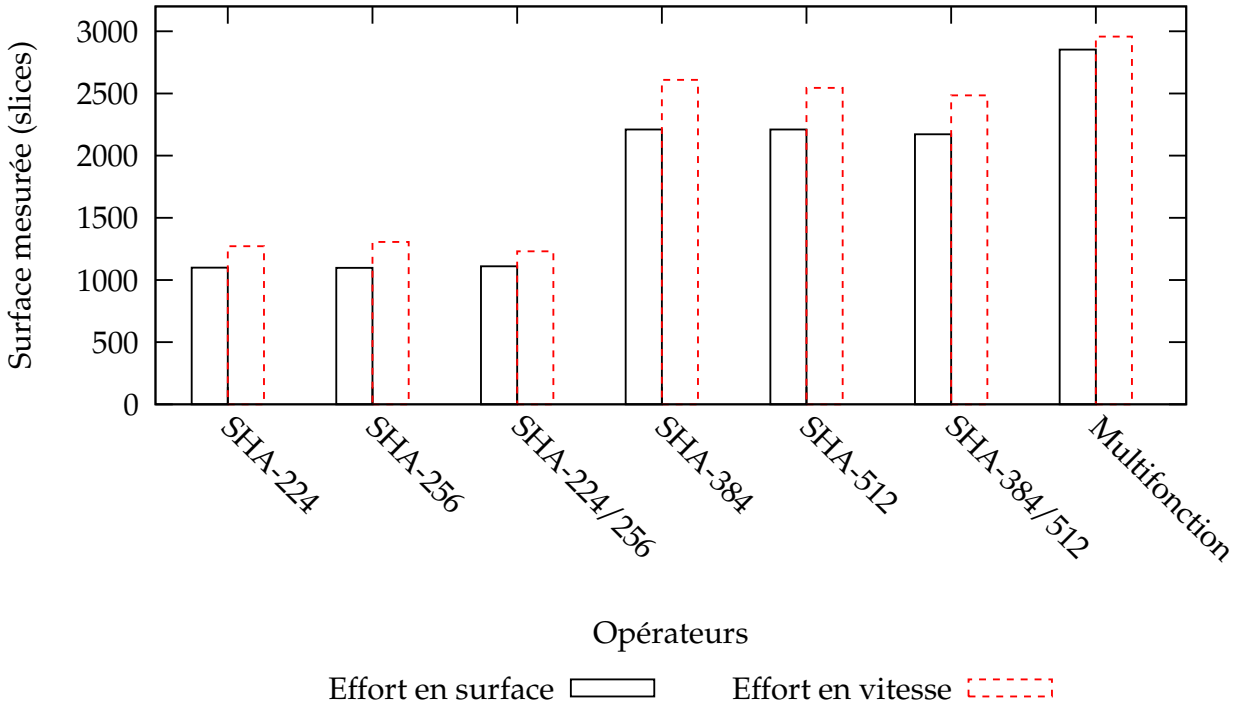

Figure 3.15 *Surface des différents opérateurs sous leur forme rapide sur Spartan3.*

de matériel, et introduit deux cycles de latence par message traité. L'analyse des résultats de placement-routage montre que la fréquence d'horloge sur Spartan en est accrue en moyenne de 7% pour les fonctions 32 bits, de 17% pour les fonctions 64 bits, et de 22% pour l'opérateur multifonction. Sur Virtex, la fréquence est accrue en moyenne de 10 % pour les fonctions 32 bits, de 6% pour les fonctions 64 bits, et de 9% pour l'opérateur multifonction.

Analyse du débit maximum

La fréquence d'horloge a un effet linéaire sur le débit. La latence de deux cycles qu'entraîne l'amélioration proposée peut être négligée dans le calcul du débit. En effet elle n'est que de deux cycles par message, et dans un cadre d'utilisation correct des fonctions de la norme, la taille des messages est très supérieure à celle d'un bloc.

Les débits maximum pour Spartan 3, avec une optimisation en vitesse, sont de 280 Mbit/s pour SHA-256, 409 Mbit/s pour SHA-512 et 440 Mbit/s ou 350 Mbit/s pour l'architecture multifonction, selon si on considère 2 fonctions SHA-256 concurrentes ou une fonction SHA-512. Dans les mêmes conditions, les débits correspondants sur Virtex sont de 308 Mbit/s pour SHA-256, 441 Mbit/s pour SHA-512 et 400 ou 320 Mbit/s pour l'architecture multifonction. Dans la mesure où les opérateurs SHA-224 et SHA-256 d'une part, SHA-384 et SHA-512 d'autre part sont identiques, les ratio entre leurs débits respectifs est simplement celui des tailles d'empreinte qu'ils génèrent, c'est à dire $7/8$ et $3/4$.

Analyse du rapport Débit sur Surface

Le critère le plus représentatif de l'efficacité des opérateurs est le ratio débit sur surface, qui prend en compte le coût matériel et la vitesse de calcul. Il est ici mesuré en Mbit par seconde et par

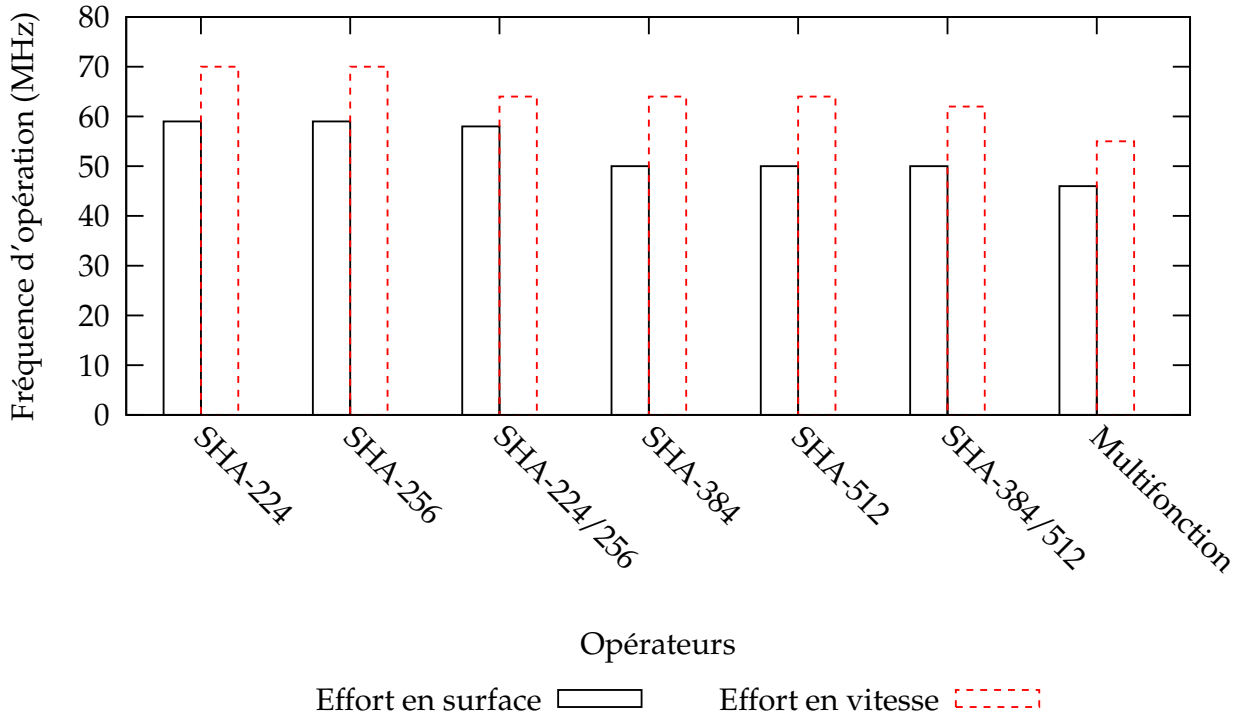

Figure 3.16 *Fréquence de fonctionnement des opérateurs sous leur forme rapide sur Spartan3.*

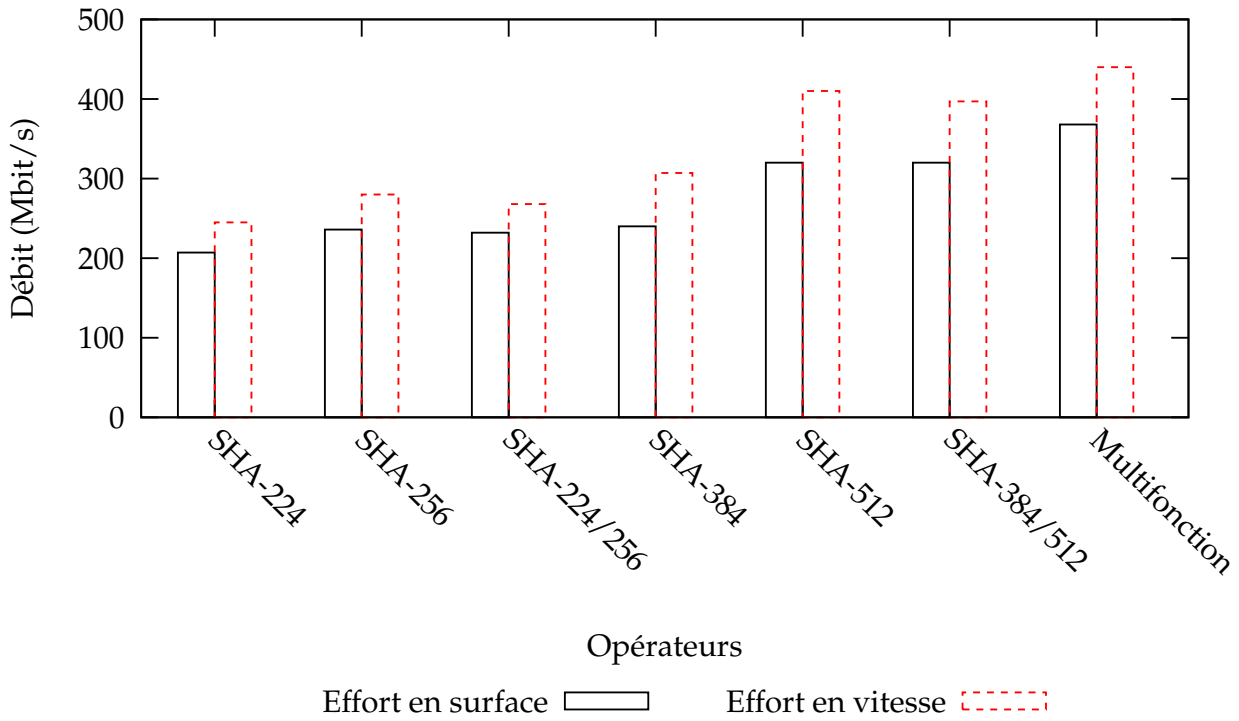

Figure 3.17 *Débit maximum des opérateurs sous leur forme rapide sur Spartan3.*

tranche. Il va permettre une exploration des compromis vitesse/surface. La différence d'efficacité entre les opérateurs SHA-256 et SHA-512 et leurs variantes 224 et 384 est constante, puisque les ressources et fréquence de calcul sont presque identiques.

L'efficacité de SHA-256 avec l'amélioration du chemin critique est de 21% contre 17% sur

Architectures de référence	Surface (Tranches)	Fréquence (MHz)	Cycles par bloc	Débit (Mbit/s)	Débit/Surface (Mbit/s/tranche)
[55] SHA-256	*2120	83	81	262	0.123
[55] SHA-384	*3932	74	97	293	0.075
[55] SHA-512	*4474	75	97	396	0.089
[56] SHA-384/512	*5828	38	pipe-line	479	0.082
[55] SHA-256/384/512	*4768	74	81/97	233.9/390.6	0.049/0.082
Architectures proposées					
SHA-224	1297	77	64(+2)	269.5	0.208
SHA-256	1306	77	64(+2)	308	0.236
SHA-224/256	1260	69	64(+2)	276	0.219
SHA-384	2581	69	80(+2)	331	0.128
SHA-512	2545	69	80(+2)	442	0.174
SHA-384/512	2573	66	80(+2)	422	0.164
****SHA-2 Multi-mode**	2951	50	64/80(+2)	2×200/320	0.136/0.108

*1 CLB=2 tranches pour Virtex, cible :Virtex 200/400XCV,
**Le débit maximum en multifonction est de 400 Mbit/s pour 2×32 bits,
et de 320 Mbit/s pour 64 bits

Table 3.11 *Résultats de synthèse et comparaisons avec les architectures précédentes.*

Spartan3 avec effort en vitesse, et de 21% contre 18% avec effort en surface. Elle est de 16% contre 13%, et de 14% contre 13% pour SHA-512. Enfin pour l'architecture multifonction en 2×32 bits, elle est de 15% contre 11% et de 13% contre 11% ; dans le cas d'un calcul 64 bits, elle est de 12% contre 9% et 10% contre 9%. L'amélioration proposée augmente l'efficacité dans tous les cas (pour le critère d'efficacité proposé). Si les ressources disponibles le permettent, il sera toujours plus efficace d'appliquer le découpage du chemin critique.

3.6.4 Opérateur concurrent multifonction

L'architecture proposée pour l'opérateur multifonction est nouvelle en ce sens qu'elle offre une plus grande flexibilité, en permettant non seulement le calcul de toutes les fonctions de la norme SHA-2 dans un seul composant, mais aussi en optimisant l'utilisation du matériel dans ce composant. Elle permet en effet à deux calculs SHA-224 ou SHA-256 de s'exécuter de manière concurrente, comme s'ils se faisaient dans des opérateurs indépendants (et pas seulement en parallèle). Cette flexibilité ne requiert que peu de ressources supplémentaires par rapport à celles qu'on trouve dans un opérateur SHA-512, car une grande partie du matériel est commun à tous les modes de calcul.

Dans le cadre d'un système où toutes les fonctions de hachage de la famille SHA-2 sont requises, il est clair que l'opérateur multifonction est plus efficace en termes de surface. Sur un Spartan3, pour un effort en vitesse, les architectures séparées demandent 25% de surface supplémentaire par rapport à l'opérateur multifonction, et 15% pour un effort en surface. En termes de débit, l'architecture reconfigurable offre des performances proches des implantations séparées. Au regard du critère débit sur surface, les architectures séparées optimisées présentées ici restent plus efficaces.

3.6.5 Comparaisons avec les architectures existantes

La comparaison des architectures proposées est faite avec d'autres implantations de fonctions séparées, et une architecture multifonction [55, 56]. (voir le tableau 3.11)

Le but de [55] était d'implanter SHA-256, 384 et 512 dans un unique opérateur avec pour cible le Virtex XCV200. Lors d'un calcul de la fonction SHA-256, la moitié du chemin de donnée (donc du matériel) reste inutilisée. La référence [56] étudie une architecture pipe-line ASIC pour SHA-384/512. Dans les deux travaux, de 16 (pour $w = 32$ bits) à 32 (pour $= 64$ bits) cycles sont exigés pour acquérir et remplir chaque bloc du message avant que ce bloc ne soit haché. À ceci s'ajoute un cycle pour le contrôle. Ces cycles supplémentaires sont évités dans le système proposé ici, qui grâce un traitement « au vol » des données demande 25% de cycles en moins que [56] pour traiter chaque bloc. De plus, grâce au pré-calcul les fréquences d'opération obtenues sont supérieures à celles de [56] et proches de [55], pour un coût matériel très inférieur.

Dans le cas des opérateurs séparés, nous obtenons un débit supérieur aux implantations de [55] de 10 à 18%, avec une réduction de la surface des opérateurs allant de 40 à plus de 45% (nous n'avons pas pris en compte l'opérateur SHA-384). D'autre part, bien que leur débit soit de 8 à 12% inférieur à celui de l'implantation pipe-line de [56], nos opérateurs occupent moins de 45% de sa surface. En conséquence, l'efficacité de nos opérateurs (le ratio débit par tranche) est double.

De même, l'opérateur multifonction emploie considérablement moins de ressources que l'architecture 256/384/512 présentée dans [55], avec 2951 tranches contre 4768 tranches, soit presque 40% plus petit. Du fait de sa fréquence d'opération moindre sur Virtex, notre opérateur multifonction obtient un débit plus faible pour les opérations 64 bits, avec 320 Mbit/s contre 390 Mbit/s. De même en ne calculant qu'une fonction 32 bits, le débit est de 200 Mbit/s contre 234. Cependant ce débit peut être doublé, soit 400 Mbit/s en faisant calculer deux fonctions 32 bits en parallèle dans le même composant. L'efficacité de notre opérateur est de $2 \times 0.068 = 0.136$ Mbit/s/tranche en mode 2×32 bits, ou 0.108 Mbit/s/tranche en mode 64 bits, contre 0.049 et 0.082 Mbit/s/tranche pour [55], soit presque 40% de mieux pour une seule fonction 32 bits, 177% pour deux fonctions 32 bits concurrentes et plus de 30% pour une fonction 64 bits !

3.7 Conclusion

Dans ce chapitre, nous avons présenté des opérateurs pour la famille SHA-2 qui optimisent l'utilisation du chemin de données et éliminent toutes les latences inutiles. L'architecture à plusieurs modes proposée peut exécuter soit une fonction SHA-384 ou SHA-512, ou se comporter comme deux opérateurs indépendants calculant des fonctions SHA-224 ou SHA-256, et ce avec un surcoût matériel minimal. Nous avons démontré l'avantage d'intégrer un mode 32 bits concurrent quand une fonction de hachage 64 bits est requise.

De plus, la nouvelle architecture réalise des performances comparables aux réalisations précédentes tout en exigeant beaucoup moins de matériel. Et surtout, les architectures présentées sont plus efficaces vis à vis du rapport débit sur surface.

3.8 Remerciements

Ces travaux ont été en partie financés grâce à l'iCORE (Informatics Circle of Research Excellence), le NSERC (Natural Sciences and Engineering Research Council of Canada), le CMC et

une bourse ACI du Ministère français délégué à l'Enseignement supérieur et à la Recherche.

Conclusion

Les différents travaux réalisés dans le cadre de cette thèse ont permis d'obtenir des implantations efficaces d'opérateurs arithmétiques dédiés à des applications spécifiques sur FPGA. Des générateurs automatiques ont été développés qui servent à écrire et valider facilement des opérateurs adaptés à de nombreuses cibles et utilisations. Je détaille les résultats pour chaque chapitre, en y ajoutant les perspectives de développements futurs.

Nous avons travaillé sur des opérateurs évaluant un ensemble de fonctions à l'aide d'approximations polynomiales. Un algorithme implanté dans un programme C++ de plus de 5000 lignes fournit des approximations avec des coefficients creux et où des puissances sont tronquées pour minimiser leur coût matériel. Un grande partie du matériel est partagée entre l'évaluation des différentes fonctions, ce qui permet d'en faire une solution bien plus intéressante qu'une implantation séparée des opérateurs. Le choix du langage C++ et une optimisation poussée de notre programme rendent possible une exploration vaste de l'espace des paramètres. La version initiale de cette méthode a été présentée lors de la *16ème International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, à Samos en 2005 [3]. Elle y a remporté le prix du meilleur papier. La méthode de génération des opérateurs spécialisés dans le calcul des puissances utilisés par nos approximations a été présentée lors de la conférence internationale *Advanced Signal Processing Algorithms, Architectures and Implementations XVI*, à San Diego en 2006 [4]. Un article de journal en cours de finalisation présente l'extension de cette méthode décrite dans le premier chapitre. Le programme pour la génération automatique d'opérateurs d'évaluation sera prochainement mis à la disposition de la communauté arithmétique.

Le logiciel `Divgen`, présenté lors de la conférence internationale *Advanced Signal Processing Algorithms, Architectures and Implementations XV* à San Diego en 2005 [5], permet d'obtenir facilement les descriptions VHDL de diviseurs optimisés. Ces diviseurs sont générés pour différents algorithmes et options passés en paramètres. Il s'agit d'un programme C++ de 5000 lignes environ. Une extension vers d'autres fonctions algébriques est implantée (en partie) par un programme Maple de 2000 lignes environ. Ce programme permet d'obtenir des opérateurs à récurrence de chiffres très efficaces grâce à un calcul fin des bornes et des erreurs. En effet, il peut manipuler des termes qu'il est difficile de gérer lors d'une démonstration manuelle. De plus, une étude complète faite à la main pour un jeu de paramètres donné est souvent fastidieuse et sujette à de nombreuses erreurs. La génération automatique de descriptions VHDL d'opérateurs à additions et décalages optimisés et validés numériquement fait donc de ces opérateurs une solution intéressante pour des non spécialistes.

En septembre et octobre 2005, dans le cadre d'une collaboration entre le LIP et le laboratoire ATIPS de l'Université de Calgary, j'ai eu l'occasion de travailler avec Ryan Glabb, alors en thèse à l'ATIPS. Nous avons réalisé une implantation améliorée des fonctions de hachage cryptographique de la norme SHA-2, avec des besoins matériel réduits. Nous proposons aussi une version fonction-

nant à haute fréquence. Finalement nous avons obtenu une architecture multifonction capable de calculer toutes les fonctions de la norme, et se comporter comme deux opérateurs SHA-224 ou SHA-256 indépendants. Cette capacité en fait une solution très intéressante pour une implantation matérielle de la norme. Ces travaux ont donné lieu à un article présenté lors de la *2006 International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA 2006)* [6], et à un article plus détaillé qui est paru dans le numéro spécial *Embedded Cryptographic Hardware* du journal *Journal of Systems Architecture* cette année [7]. De nombreuses fonctions cryptographiques peuvent bénéficier d'une implantation matérielle dans un opérateur spécialisé. Il est aussi possible de calculer certaines familles de fonctions cryptographiques dans une architecture partagée, avec des performances proches d'implantations séparées, pour un coût matériel bien moindre.

Perspectives

D'ici la fin de l'année, nous pensons mettre en ligne notre programme de génération d'approximations polynomiales. Il devrait par la suite être étendu et amélioré pour autoriser d'autres types de schémas d'évaluation, de découpages d'intervalles, et une exploration plus complète de l'espace des paramètres. Un article de journal est en cours de rédaction.

Une nouvelle version du générateur `Divgen`, étendue à d'autres fonctions calculables par les algorithmes SRT, est en cours d'écriture. Il nous reste à étudier l'effet de certaines techniques d'optimisation, comme le repli des tables ou la saturation. D'autres variations des algorithmes SRT (schémas récursifs, avec prédiction ou par arrondi par exemple) [51] restent à ajouter. Nous comptons aussi adapter ce générateur à des cibles ASIC. Cette extension de l'outil `Divgen` sera présentée en détail dans un rapport de recherche.

J'envisage de partir pour un séjour post-doctoral à l'Université de Calgary au Canada à la fin de ma thèse. J'espère pouvoir mettre à profit mon séjour pour approfondir l'étude d'opérateurs arithmétiques matériels dédiés à la cryptographie, et continuer mes travaux sur les opérateurs SRT.

3.9 Publications personnelles

[1] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Évaluation de polynômes et de fractions rationnelles sur FPGA avec des opérateurs à base d'additions et décalages en grande base. *Dans 10e SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 85–96, avril 2005.

[2] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Étude statistique de l'activité de la fonction de sélection dans l'algorithe de E-méthode. *Dans 5e journées d'études Faible Tension Faible Consommation (FTFC)*, pages 61–65, mai 2005.

[3] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Small FPGA polynomial approximations with 3-bit coefficients and low-precision estimations of the powers of x. *Dans 16th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 334–339. IEEE Computer Society, juillet 2005.

[4] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : New identities and transformations for hardware power operators. *Dans Advanced Signal Processing Algorithms, Architectures and Implementations XVI*. SPIE, août 2006.

[5] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Divgen : a divider unit generator. *Dans Advanced Signal Processing Algorithms, Architectures and Implementations XV*. SPIE, août 2005.

[6] R. GLABB, L. IMBERT, G. A. JULLIEN, A. TISSERAND et N. VEYRAT-CHARVILLON : Multi-mode operator for SHA-2 hash functions. *Dans 2006 International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA 2006)*, pages 207–210, 2006.

[7] R. GLABB, L. IMBERT, G. A. JULLIEN, A. TISSERAND et N. VEYRAT-CHARVILLON : Multi-mode operator for SHA-2 hash functions. *Journal of Systems Architecture*, 53(2-3):127–138, 2007.

[8] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. *Dans 11e SYMPosium en Architectures nouvelles de machines (SYMPA)*, pages 130–141, octobre 2006.

[9] R. MICHARD, A. TISSERAND et N. VEYRAT-CHARVILLON : Carry prediction and selection for truncated multiplication. *Dans Workshop on Signal Processing Systems (SiPS'06)*. IEEE Computer Society, octobre 2006.

3.10 Références générales

[10] M. D. ERCEGOVAC : *A general method for evaluation of functions and computations in a digital computer.* Thèse de doctorat, University of Illinois at Urbana-Champaign, 1975.

[11] M. D. ERCEGOVAC et T. LANG : *Digital Arithmetic.* Morgan Kaufmann, 2003.

[12] J. M. MULLER : *Elementary Functions : Algorithms and Implementation.* Birkhäuser, 2e édition, 2006.

[13] B. PARHAMI : *Computer Arithmetic : Algorithms and Hardware Designs.* Oxford University Press, 2000.

[14] N. BRISEBARRE, J. M. MULLER et A. TISSERAND : Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, 32(2):236–256, juin 2006.

[15] W.P. BURLESON : Polynomial evaluation in VLSI using distributed arithmetic. *IEEE Transactions on Circuits and Systems*, 37(10):1299–1304, octobre 1990.

[16] F. de DINECHIN et A. TISSERAND : Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, mars 2005.

[17] J. DUPRAT et J. M. MULLER : Hardwired polynomial evaluation. *Journal of Parallel and Distributed Computing*, 5(3):291–309, juin 1988.

[18] M. D. ERCEGOVAC, T. LANG, J. M. MULLER et A. TISSERAND : Reciprocation, square root, inverse square root, and some elementary functions using small multipliers. *IEEE Transactions on Computers*, 49(7):628–637, juillet 2000.

[19] E. REMES : Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *Comptes Rendus de l'Académie des Sciences de Paris*, 198: 2063–2065, 1934.

[20] M. SCHULTE et J. STINE : Approximating elementary functions with symmetric bipartite tables. *IEEE Transactions on Computers*, 48(8):842–847, août 1999.

[21] N. TAKAGI : Powering by a table look-up and a multiplication with operand modification. *IEEE Transactions on Computers*, 47(11):1216–1222, novembre 1998.

[22] N. J. HIGHAM : *Handbook of Writing for the Mathematical Sciences.* Society for Industrial and Applied Mathematics, 2^{e} édition, 1998.

[23] D. E. KNUTH : *The Art of Computer Programming, Volume III : Sorting and Searching.* Addison-Wesley, 1973.

[24] P. MARKSTEIN : *IA-64 and Elementary Functions : Speed and Precision.* Hewlett-Packard Professional Books. Prentice Hall, 2000.

[25] W. H. PRESS, S. A. TEUKOLSKY, W. T. VETTERLING et B. P. FLANNERY : *Numerical Recipes in C : The Art of Scientific Computing.* Cambridge University Press, 1992.

[26] J. DETREY et F. de DINECHIN : Second order function approximation using a single multiplication on FPGAs. *Dans 14th International Conference on Field-Programmable Logic and Applications (FPL)*, numéro 3203 de LNCS, pages 221–230. Springer, septembre 2004.

[27] J. DETREY et F. de DINECHIN : Table-based polynomials for fast hardware function evaluation. *Dans 16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 328–333. IEEE Computer Society, juillet 2005.

[28] H. HASSLER et N. TAKAGI : Function evaluation by table look-up and addition. *Dans 12th IEEE Symposium on Computer Arithmetic (ARITH12)*, pages 10–16. IEEE Computer Society, juillet 1995.

[29] K. JOHANSSON, O. GUSTAFSSON et L. WANHAMMAR : Approximation of elementary functions using a weighted sum of bit-products. *Dans IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 795–798. IEEE Computer Society, mai 2006.

[30] E. J. KING et E. E. SWARTZLANDER : Data-dependent truncation scheme for parallel multipliers. *Dans 31th Asilomar Conference on Signals, Systems & Computers*, volume 2, pages 1178–1182. IEEE Computer Society, novembre 1997.

[31] A. A. LIDDICOAT et M. J. FLYNN : Parallel square and cube computations. *Dans 34th Asilomar Conference on Signals, Systems, and Computers*, pages 1325–1329. IEEE Computer Society, octobre 2000.

[32] J. A. PINEIRO, J. D. BRUGUERA et J. M. MULLER : Faithful powering computation using table look-up and a fused accumulation tree. *Dans 15th IEEE Symposium on Computer Arithmetic (ARITH15)*, pages 40–47. IEEE Computer Society, 2001.

[33] A. J. AL-KHALILI et A. HU : Design of a 32-bit squarer — exploiting addition redundancy. *Dans IEEE International Symposium on Circuits and Systems (ISCAS)*, volume V, pages 325–328. IEEE Computer Society, mai 2003.

[34] P. IENNE et M. A. VIREDAZ : Bit-serial multipliers and squarers. *IEEE Transactions on Computers*, 43(12):1445–1450, décembre 1994.

[35] S. KRITHIVASAN, M. J. SCHULTE et J. GLOSSNER : A subword-parallel multiplication and sum-of-squares unit. *Dans IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 19–20. IEEE Computer Society, février 2004.

[36] B. R. LEE et N. BURGESS : Improved small multiplier based multiplication, squaring and division. *Dans 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 91–97. IEEE Computer Society, avril 2003.

[37] E. G. WALTERS et M. J. SCHULTE : Efficient function approximation using truncated multipliers and squarers. *Dans 17th IEEE Symposium on Computer Arithmetic (ARITH17)*, pages 232–239. IEEE Computer Society, juin 2005.

[38] C. L. WEY et M. D. SHIEH : Design of a high-speed square generator. *IEEE Transactions on Computers*, 47(9):1021–1026, septembre 1998.

[39] K. E. WIRES, M. J. SCHULTE, L. P. MARQUETTE et P. I. BALZOLA : Combined unsigned and two's complement squarers. *Dans 33th Asilomar Conference on Signals, Systems & Computers*, volume 2, pages 1215–1219. IEEE Computer Society, octobre 1999.

[40] A. EDELMAN : The mathematics of the pentium division bug. *SIAM Review*, 39(1):54–67, mars 1997.

[41] M. D. ERCEGOVAC et T. LANG : On-the-fly conversion of redundant into conventional representations. *IEEE Transactions on Computers*, 36(7):895–897, juillet 1987.

[42] M. D. ERCEGOVAC et T. LANG : On-the-fly rounding. *IEEE Transactions on Computers*, 41(12):1497–1503, décembre 1992.

[43] P. KORNERUP : Digit selection for SRT division and square root. *IEEE Transactions on Computers*, 54(3):294–303, mars 2005.

[44] T. LANG et E. ANTELO : Radix-4 reciprocal square-root and its combination with division and square root. *IEEE Transactions on Computers*, 52(9):1100–1114, 2003.

[45] S. F. OBERMAN et M. J. FLYNN : Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, février 1997.

[46] S. F. OBERMAN et M. J. FLYNN : Division algorithms and implementations. *IEEE Transactions on Computers*, 46(8):833–854, août 1997.

[47] S. F. OBERMAN et M. J. FLYNN : Minimizing the complexity of SRT tables. *IEEE Transactions on VLSI systems*, 6(1):141–149, mars 1998.

[48] J. E. ROBERTSON : A new class of division methods. *IRE Transactions Electronic Computers*, EC-7:218–222, septembre 1958.

[49] N. TAKAGI et S. KUWAHARA : A VLSI algorithm for computing the euclidean norm of a 3d vector. *IEEE Transactions on Computers*, 49(10):1074–1082, octobre 2000.

[50] K. D. TOCHER : Techniques of multiplication and division for automatic binary computers. *Quarterly Journal of Mechanics and Applied Mathematics*, 11-part 3:368–384, 1958.

[51] M. D. ERCEGOVAC et T. LANG : *Division and Square Root : Digit-Recurrence Algorithms and Implementations.* Kluwer Academic Publishers, Boston, 1994.

[52] J. FANDRIANTO : Algorithms for high speed shared radix 4 division and radix 4 square root. *Dans 8th IEEE Symposium on Computer Arithmetic (ARITH8)*, pages 73–79. IEEE Computer Society, 1987.

[53] N. TAKAGI : A hardware algorithm for computing reciprocal square root. *Dans 15th IEEE Symposium on Computer Arithmetic (ARITH15)*, pages 94–100. IEEE Computer Society, 2001.

[54] R. B. LEE, Z. SHI et X. YANG : Cryptography efficient permutation instructions for fast software. *IEEE Micro*, 21(6):56–69, 2001.

[55] N. SKLAVOS et O. KOUFOPAVLOU : Implementation of the SHA-2 hash family standard using FPGAs. *The Journal of Supercomputing*, 31(3):227–248, mars 2005.

[56] M. MCLOONE et J. MCCANNY : Efficient single-chip implementation of SHA-384 & SHA-512. *IEEE International Conference on Field-Programmable Technology (FTP)*, pages 311–314, 2002.

[57] X. WANG, Y. L. YIN et H. YU : Finding collisions in the full SHA-1. *Lecture Notes in Computer Science*, 3621:17–36, 2005.

[58] J. DAEMEN et V. RIJMEN : *The Design of Rijndael : AES - The Advanced Encryption Standard.* Springer, 2002.

[59] A. J. MENEZES, P. C. van OORSCHOT et S. A. VANSTONE : *Handbook of Applied Cryptography.* CRC Press, 1997.

[60] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY : *FIPS PUB 180-1 : Secure Hash Standard.* NIST, avril 1995.

[61] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY : *FIPS PUB 180-2 : Secure Hash Standard.* NIST, février 2004.

[62] R. ANDERSON et E. BIHAM : Tiger : A fast new hash function. *Dans IWFSE : International Workshop on Fast Software Encryption, LNCS.* Springer, 1996.

[63] H. DOBBERTIN, A. BOSSELAERS et B. PRENEEL : RIPEMD-160 : A strengthened version of RIPEMD. *Dans IWFSE : International Workshop on Fast Software Encryption, LNCS.* Springer, 1996.

[64] V. RIJMEN et P. S. L. M. BARRETO : The WHIRLPOOL hash function. document World-Wide Web, 2001.

[65] R. L. RIVEST : The MD5 message-digest algorithm. Internet informational RFC 1321, avril 1992.

[66] X. WANG, D. FENG, X. LAI et H. YU : Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004.